Investigation of Service Selection Algorithms for Grid Services

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Division of Computer Science
University of Saskatchewan
Saskatoon

By
Tapashree Guha

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Grid computing has emerged as a global platform to support organizations for coordinated sharing of distributed data, applications, and processes. Additionally, Grid computing has also leveraged web services to define standard interfaces for grid services adopting the service-oriented view. Consequently, there have been significant efforts to enable applications capable of tackling computationally intensive problems as services on the Grid. In order to ensure that the available services are assigned to the high volume of incoming requests efficiently, it is important to have a robust service selection algorithm. The selection algorithm should not only increase access to the distributed services, promoting operational flexibility and collaboration, but should also allow service providers to scale efficiently to meet a variety of demands while adhering to certain current Quality of Service (QoS) standards. In this research, two service selection algorithms, namely the Particle Swarm Intelligence based Service Selection Algorithm (PSI Selection Algorithm) based on the Multiple Objective Particle Swarm Optimization algorithm using Crowding Distance technique, and the Constraint Satisfaction based Selection (CSS) algorithm, are proposed. The proposed selection algorithms are designed to achieve the following goals: handling large number of incoming requests simultaneously; achieving high match scores in the case of competitive matching of similar types of incoming requests; assigning each services efficiently to all the incoming requests; providing requesters the flexibility to provide multiple service selection criteria based on a QoS metric; selecting the appropriate services for the incoming requests within a reasonable time. Next, the two algorithms are verified by a standard assignment problem algorithm called the Munkres algorithm. The feasibility and the accuracy of the proposed algorithms are then tested using various evaluation methods. These evaluations are based on various real world scenarios to check the accuracy of the algorithm, which is primarily based on how closely the requests are being matched to the available services based on the QoS parameters provided by the requesters.

Acknowledgements

In the first place I would like to record my gratitude to Dr. Simone Ludwig, my supervisor, for her supervision, advice, and guidance from the very early stage of this research, and in giving me extraordinary experiences throughout the work. Above all and the most needed, she provided me unflinching encouragement and support in various ways. I gratefully acknowledge her crucial contribution, and her encouragement that has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come. I am grateful in every possible way to her for this. I am obliged to, Dr. Kiel, and Dr. McQuillan, my committee members, for agreeing to review my thesis and give valuable suggestions on it. I thank my colleagues, and department staff for their support and encouragement in making this research a success. I would also like to extend a special appreciation towards my family for their love, valuable advices and blessings. Finally, I would like to thank everybody else who was important to the successful realization of thesis as well as express my apology for them whose name I could not mention personally one by one.

Dedicated to my family.

Contents

List of Tables

List of Figures

List of Abbreviations

AO	Actual Organization
ACO	Ant Colony Optimization
EA	Evolutionary Algorithm
CD	Crowding Distance
CSS	Constraints Satisfaction Based Service Selection Algorithm
DNA	Deoxyribonucleic Acid
CPU	Central Processing Unit
GA	Genetic Algorithm
GRIS	Grid Index Services
ICT	Information and Communication Technology
MOA	Multi Objective Algorithms
MOEA	Multi Objective Evolutionary Algorithm
MOOGA	Multi Objective Optimization Genetic Algorithm
MOPSO	Multiple Objective Particle Swarm Optimization
MS	Match Score
NSGA-II	Non-Dominated Sorting Algorithm-II
NPGA	Niched Pareto Genetic Algorithm
PSO	Particle Swarm Optimization
PSI	Particle Swarm Intelligence Based Selection Algorithm
QoS	Quality of Service
RAM	Random Access Memory
URL	Uniform Resource Locator
VEGA	Vector Evaluated Genetic Algorithm
VO	Virtual Organization
VOMS	Virtual Organization Membership Services

Chapter 1

Introduction

Since the beginning of the computer era, the primary concerns of computer scientists have centered on increasing the scope of its application areas, and also to increase the performance and efficiency of the same. Each success of developing a faster and efficient computer has led to the desire of finding solutions for larger and more sophisticated problems, which in turn led to the enhancement of computer models to reflect higher level of computational efficiency. Additionally, with the increase in complexities and scaling of domain problems in science, engineering and other relevant areas, the last few decades have seen a proportionate increase in the need for high performance computing and its applications to solve and model these problems. For example, in research areas like fluid dynamics, molecular dynamics, global climate modeling, etc., the researchers are depending on high performance applications to explore and simulate interesting phenomena in these fields [2]. Due to the popularity of high-speed networks, and the World Wide Web, there have been significant efforts to provide these computationally expensive applications as services on a standard platform such as the web, or the Grid. These services are accessed remotely across the network by the users, and which not only promotes seamless execution of the requests submitted by the users, but also enables sharing of these services amongst various other geographically dispersed users. Few examples of high performance services that are available online is data mining, theorem proving and logic, parallel numerical computations, etc. For highly on-demand services, it will often be the case that the services are replicated at multiple sites (referred to as service providers), so that in a situation where there is a large number of incoming requests for the services, all the service providers together can serve the client requests. In order to achieve the goals of executing the client requests remotely, efficiently and to dynamically tackle the high volume of user requests for these services, the need for high-performance computational resources deployed in the high speed networks becomes obligatory. The efforts to address such new requirements, forced computer scientists to improve the efficiency of the computer models, which led to the invention of high performance and distributed computing. Significant changes in the underlying architecture and the computational speed of the computer models were made, to support the execution of computationally intensive applications simultaneously. The changes were accompanied by both the increase in the efficiency and the complexity of the underlying software of the high-speed networks. Nevertheless,

there were still problems, in the fields of science, engineering, and business, which were not effectively dealt with using supercomputers. The problems at hand, due to their size and complexity, often required a variety of heterogeneous resources that are not available on a single machine [3]. Hence as a solution, in the late nineties, a team of scientists had conducted an experimental study on the collaborative and simultaneous use of geographically distributed resources, to act as a single powerful computer. This new approach was initially known by several names, such as metacomputing, scalable computing, global computing, Internet computing, and more recently it has been called peer-to-peer or Grid computing [4]. The initial efforts in Grid computing started as a project to link supercomputing sites, but now the Grid has rapidly emerged as a dominant paradigm for wide area high-performance distributed computing. It is a special type of parallel computing which relies on computers connected to a high-speed network by a conventional network interface, such as Ethernet, as opposed to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus. The primary advantage of the Grid computing paradigm is that, each node can be used as a resource, which when combined can produce similar computing resources to a multiprocessor supercomputer, but at lower cost [5]. The geographically dispersed nature of the Grid is generally favorable, to run multiple parallel computations at the same time. This might be a different simulation for the same project, or computations for completely different applications. Hence, the basic purpose of the Grid is to provide a service-oriented infrastructure that uses standardized protocols, and to allow users to have seamless access, and to share geographically distributed computing resources providing various services. Due to these features, many applications can benefit from the Grid infrastructure, including collaborative engineering, data exploration, high-throughput computing, and of course distributed supercomputing [4]. A detailed description of the Grid and the virtual organization concept is given in the following sections.

1.1 The Grid: Definition and Related Concepts

The term "Grid", as shown in Figure 1.1 [6], was first proposed in the mid 1990s, and is mainly defined as a distributed computing infrastructure for advanced science and engineering [5]. The first formal definition of the"Grid" was proposed by Carl Kesselman and Ian Foster, in their book [7], "The Grid: Blueprint for a New Computing Infrastructure.", 1998. They defined the Grid as "a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." They have also explained that, the Grid can be distinguished from the conventional distributed computing because it focuses on the large-scale resource sharing, innovative applications, and mostly conforms to the high performance orientation. It is based on concepts motivated by real and specific problems and there is a well

defined Grid technology base that addresses significant aspect of these problems [7].

Figure 1.1: The Grid Infrastructure

In an article published in the year 2001, Foster stated that "the Grid computing is concerned with coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations." The term virtual organization (VO), as defined in [5] is primarily used for a set of individuals and/or institutions sharing the resources deployed on the Grid. The "sharing" of resources in the Grid, is not merely file exchange, but it focuses more on the direct access to computers, software, data and other resources required to solve massively computational intensive problem in an unified manner. This sharing of resources amongst various VOs is highly controlled and is established, managed and exploited based on some clearly defined protocols. The description of the virtual organization concept is given in the following paragraphs [5].

1.1.1 The Virtual Organization Concept

The key concept of the Grid is the ability to realize seamless and large-scale resource-sharing among a set of participating individuals and/or organizations - formally known as Virtual Organizations (VOs), who share a common computationally and/or data-intensive goal. To achieve this goal, the individuals and/or the organization within a VO choose to share their resources amongst themselves. Some examples of VOs given in [5] are - the application service providers, storage service providers, CPU cycle providers, and consultants engaged by a car manufacturer to perform scenario evaluation during planning for a new factory. In each of these examples, the participating organizations may be given controlled access to the computers, applications, files, data, sensors and networks (or any

other resource) of other participating members of the organization. A brief description of the same can be found in Figure 1.2.

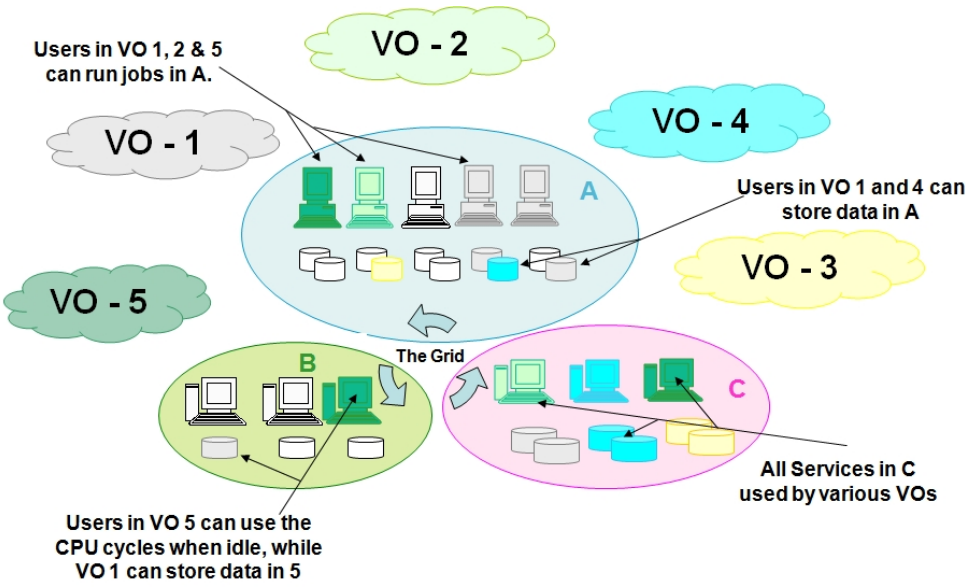

Figure 1.2: Virtual organizations sharing resources on the Grid

From Figure 1.2 [5], it can be seen that there are five virtual organizations VO-1, VO-2, ..., VO-5. In the figure apart from the VO, three Grid clusters A, B and C, informally referred to as *"actual organizations"* (AO) [5] can also be seen. These actual organizations can participate in one or more VOs by sharing some or all of its resources. For example, from the figure it can be seen that the AO C share all of its resources with VO-1,2,3,4 and 5 by allowing them to use its idle CPU cycles for processing, and to store the data, while the AOs A and B only shares some of its resources with the VOs. The resource sharing is based on rules defined by the resource owners. For example, the owner of AO B, might only allow users with less computationally intensive problems to run their requests, whereas AO A might allow massively intensive programs to be run. On the other hand, the users or the participants of the various VOs might have several other constraints. For example, the participants of the VO-5 might seek resource owners whose resources are more reliable and have a low cost. Facilitating the concept of virtual organizations is one of the most important features of Grid computing paradigm. It is important to note that, VOs are defined purely as a concept in Grid computing and not as a protocol, and the most widespread implementation of VO is the Virtual Organization Membership Service (VOMS). The VOMS basically allows the user to request a proxy-certificate from a given Virtual Organization, and then using that proxy-certificate the user may continue to submit its request(s) to a resource it needs to execute its job [8].

As derived from the above paragraphs, the key advantage of VO concept is the fact that it

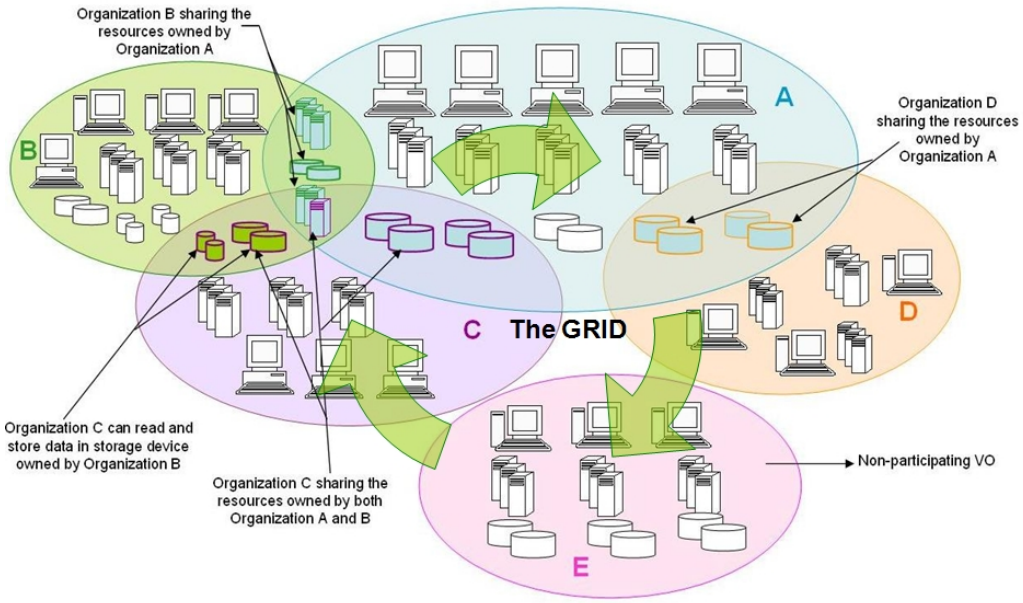

Figure 1.3: Resource sharing amongst actual organizations on the Grid

reduces the cost of the resources, as now the same resources are being shared by various participants of a single VO. The relationships of sharing these resources may vary within the AO over a period of time, depending on the availability of the resources, nature of the access permitted by each resource owner and also the participants to whom the resource has been permitted. Moreover, the resource sharing relationships amongst these resources are often not simply client server relationships, but is peer-to-peer instead. An example of the same can be seen from Figure 1.3 [5]. In order to reduce the cost and provide more flexibility to the participants of a particular organization, the AO may or may not choose to share the resources amongst themselves. For example, organization B in Figure 1.3 is sharing the resources owned by organization A, and similarly organization C is sharing the resources owned by both A and C while the organization E does not share its resources with any other organization, and hence acts as a standalone system. Therefore, the participants of certain virtual organization say VO-X, using the resources of the AO C may have the permission to access the shared resources of organizations A and B.

1.1.2 Service Selection on the Grid

Due to the advantages of VOs addressed in the previous section, the Grid computing paradigm is being rapidly deployed by industries, scientists and the engineers. It is enabling a new generation of high performance applications to be developed, to solve computationally intensive problems, and to be seamlessly accessed by the geographically dispersed users of the Grid. These applications

and the resources provided by the actual organizations are termed as *services* and, the owners of such actual organizations are termed as the *service providers*. The same services may be used in various ways, depending on the limitations placed on the access and sharing scope of the resources. However, before utilizing these services to execute a client request, a requester needs to select an appropriate service provider, from a service provider pool. Once a service provider is selected, the site will assign the appropriate number of resources to the request [9]. To illustrate this, let us consider the following scenario: A team of bioinformatics scientists from different laboratories and universities across the world need to come together to run an analysis on a set of newly generated experimental DNA sequences. But the size of the DNA sequences is huge and it is beyond the scope of the local computing system to handle it. The scientists decides to turn to Grid technologies for a solution, because of the "computationally intensive data" processing and storage capabilities of the Grid. During the analysis phase of the DNA sequences, they are able to convert their requirements into Grid compatible services (also known as *requests*) and are able to submit their query to the Grid, which is capable of analyzing a huge quantity of data. The scientists access the Grid seamlessly, meaning that they do not have any knowledge about the position of the service providers, where their requests are being executed. However, they do have specific requirements of the service providers that would execute their requests. For example, they may specify to submit their requests to one or more remote servers whose CPU average load is low and the unallocated RAM size is greater than a certain threshold.

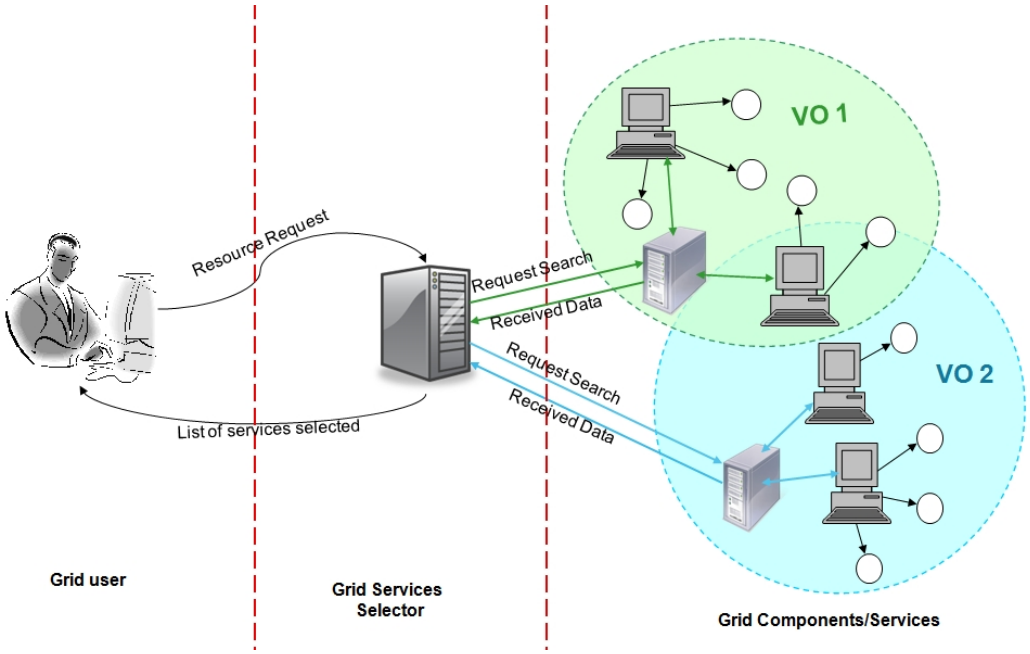

Figure 1.4: Resource sharing amongst actual organizations on the Grid

A generic case of the above scenario is explained in Figure 1.4., where the relationship between the Grid services and the Grid user is shown. A brief description of the primary procedures is given below [7]:

- At first, the Grid user summarizes the resource requirements of the Grid application to be run. The requests are then sent to the Grid service selector;
- The Grid service selector transforms the user's request into a Grid compatible search request. It then sends the search request to one or more servers in the Grid;
- Each server involved in the previous step, processes the request received from the Grid service selector. The processing of the request involves discovering the list of the available and closely matched service providers. After retrieving the data from its back end, it sends the data back to the Grid service selector;
- After collecting all data from the servers, the Grid service selector matches the available services to the corresponding request, integrates them, and then sends the matched service id(s) to the Grid user;
- Finally, the Grid user sends its request to the particular service which has been selected, and receives the results after the job has been executed.

From the above procedure, it is seen that, this service selection process necessitates an effective coordination of participating sites to handle the request(s). Most importantly it gives rise to the need of a faster and accurate selection process in order to accommodate the executions of a wide spectrum of client requests, which otherwise might lead to discrepancies in the way the requests are being served. Hence, the Grid service selector should particularly be capable of handling two primary scenarios:

1. The service selector should be able to handle multiple client requests coming in at the same time.
2. The service selector should be able to handle situations where more than one client is requesting a similar service.

To incorporate the second point, Quality of Service (QoS) parameters such as availability, reliability of the service providers etc. are required to be considered, in order to guarantee an efficient selection of services on the Grid [10, 11]. It is also mentioned in [1], for a faster and accurate selection of service providers it is required, that the service selection procedure complies with a particular QoS metric, as not meeting such criteria can impose significant impact on the accuracy of the results, which can diminish the satisfaction level of the Grid user expectations. The QoS metrics used should not only consider the network connecting these services, but should also be designed in

context of the Grid services. For an efficient and satisfactory service, a user requesting a particular service must identify a set of QoS requirements, such as the bandwidth of the network, number of processors, the time duration over which the service is required, and the expected cost the user is willing to pay, etc. At the same time, a service provider also has their capability listed in the form of QoS parameters. Among these metrics, some are decided by the service providers such as the cost, while others are provided by the third party and network managers [12]. In this research, two QoS based service selection approaches to address the issues listed above will be proposed. The first one is an enhanced version of the "classical" service selection algorithm, which uses the constraint satisfaction based problem solving approach. The second approach, uses swarm intelligence technique to overcome the drawbacks of the first algorithm, thus making it more efficient.

1.2 Thesis Organization

The rest of the thesis is structured as follows: Chapter 2 presents the problem statement and the motivation for this research, Chapter 3 presents the related work of this research, following with Chapter 4, describes the proposed approaches for service selection on the Grid - the QoS based CSS and PSI algorithms; in Chapter 5, the experimental setup, methodology and the results are discussed, after comparing and analyzing both algorithms. [1]

[1] This research work is accepted for publication in the Proceedings of the 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Dayton, Ohio, USA, November 2008.

Chapter 2

Motivation and the Problem Definition

The chapter focuses on questions such as - why do we need an efficient service selection algorithm; which are the vital application areas of the Grid needing an appropriate service selection algorithm. Furthermore, a detailed description of the problem definition is presented. Finally, the key objectives of the research is introduced in Section 2.3.

2.1 Motivation

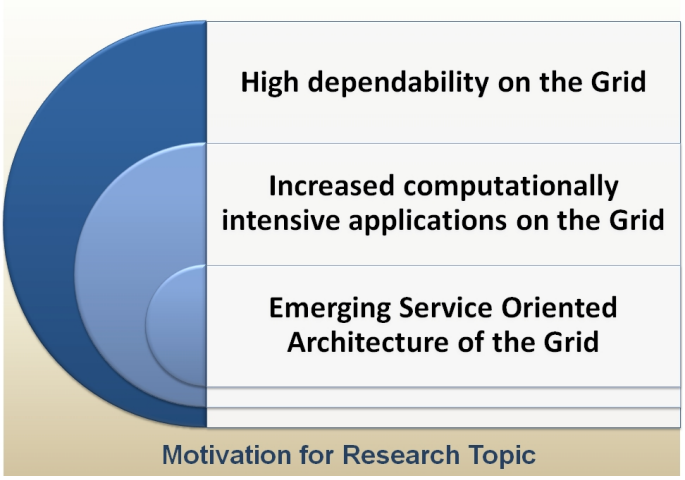

Figure 2.1: Motivation for selecting the research topic.

As seen from Figure 2.1, there are primarily three key reasons why this topic of research was chosen: the *increased dependability* on the Grid to run the *computationally intensive* applications, and finally the *emerging service oriented* architecture of the Grid. Furthermore, the Grid computing paradigm may be successfully utilized in a wide variety of application areas e.g., collaborative scientific research, financial risk analysis, product design. Grid is particularly known to be successful in managing the complexity and massiveness of these computationally intensive large-scale applications involving a lot of collaboration amongst scientists, institutions and industrial organization across the globe. These applications usually have a lot of interactions among its various

participants, and the Grid infrastructure primarily helps to connect and integrate this widely spread network, and also provides a large volume of storage, and high performance computing resources for them. By enabling dynamic sharing, abstraction and coordination of these services the Grid reduces the complexities to remotely access various services widely dispersed around the world. It also provides ways to select these services efficiently, so that the users are able to focus on their application domain without the need for considering the issues related to the integration of the distributed services. Hence, it can be seen that with the increased dependency on the Grid and its service oriented architecture to solve these massive application problems, the need for an efficient service selection algorithm has become paramount. Therefore, this research investigates the approaches to achieve efficient requests to service selection on the Grid, and also explores the performance of the algorithms in scenarios of high number of similar incoming requests, etc. [13]. To motivate this research, one of the most important application scenario of the Grid, focusing on the health care domain, (Figure 2.2 [14]) enabling collaborative research and virtualization, is presented in the following section.

2.1.1 The Healthcare Scenario

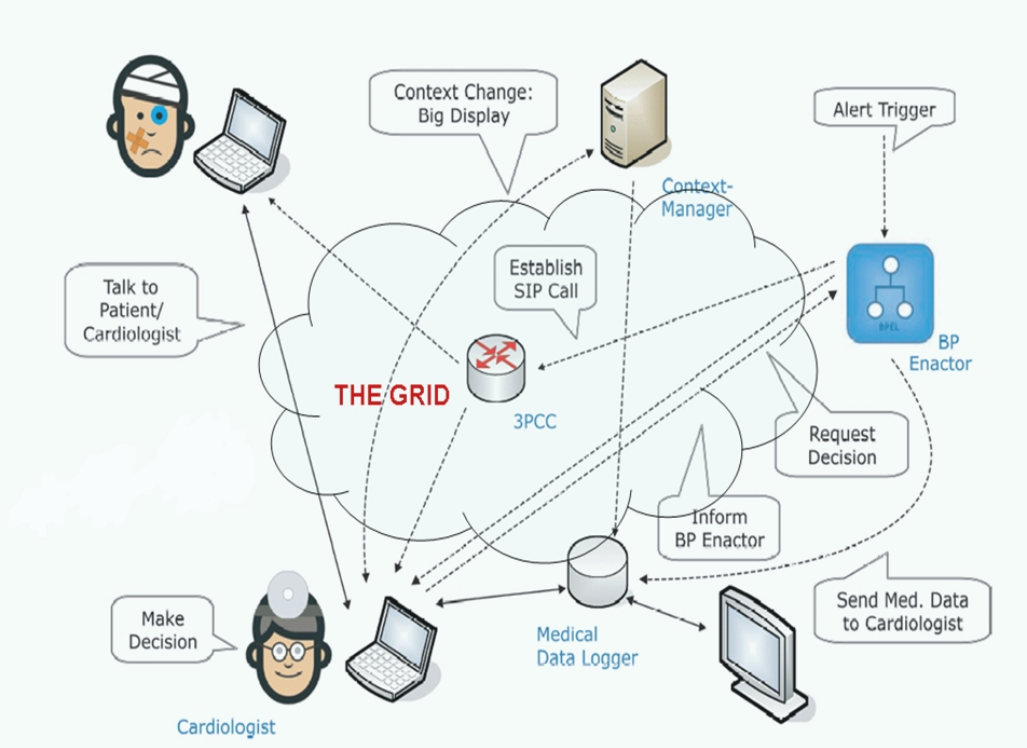

Figure 2.2: Application of the Grid in E-health

As defined in [13, 15, 16], e-Health is primarily the use of Information and Communication

Technologies (ICT) to develop an intelligent and connected environment that enables the following:

- Seamless management and understanding of citizens' healthcare across institutions in same or different countries, i.e., regardless of where the patient records are located in the system;
- Assisting health professionals to handle with collaborative challenges;
- Incorporation of the advances in health knowledge into clinical practice, i.e., to update the clinical practice system when there is an advancement in the medical knowledge base;
- Handle massive, dispersed and computationally intensive data volumes, especially in the case of genetic medicine;
- Incorporate smooth and secure handling of heterogeneous medical information across various institutions due to the lack of widely accepted standards.

The Grid paradigm is considered to be a vital approach for tackling these challenges. Many scientists along with other physicians, and medical specialists, are working collaboratively towards the creation of Health Grids. Health Grids are Grid infrastructures which comprise of applications, services and/or middleware components that deal with the specific problems arising in the processing of the biomedical data [17]. It is designed and developed to support the on-demand medical applications accessible 24 hours. These medical applications are accessed on the Grid as services which executes requests sent by the users. It is important that the requests submitted by the Health Grid user are selected and matched with appropriate services based on the requirements and the preference specified by the user. The selection process becomes complex when a large number of physicians across the world, access the services on the Grid simultaneously by submitting their requests to it. The heterogeneity of the services available, and the wide range of the incoming requests makes the selection process yet more challenging. Additionally, it might also be the case that a large number of requesters are accessing similar kind of services simultaneously, which adds another layer of complexity [13]. A detailed discussion about the service selection problem is given in the following section.

2.2 Problem Definition

A fundamental problem that the Grid research and development community is seeking to solve is: how to coordinate distributed resources amongst a dynamic set of individuals and organizations in order to solve a common collaborative goal. The problem of service selection in a Grid environment arises through the heterogeneity, distribution and sharing of the resources in different virtual organizations. The task of assigning requests to the appropriate users, efficiently also becomes challenging, because it should consider the following key factors during the selection process:

1. **Volume of the incoming request:** The service selector should be able to handle, a large number of incoming requests that are seeking to access the Grid to execute their requests simultaneously;

2. **Competitive matching:** Many requests may compete for similar kind of service(s) that in turn results in competitive matching of the requests;

3. **Optimized matching:** All the requesters in the queue, should be matched fairly, i.e., to avoid matching the requests appearing later in the queue to poor services. This usually happens if the match process is done sequentially, instead of considering all the incoming requests at the same time;

4. **Efficient:** The incoming requests should also be matched to appropriate services within a reasonable time;

5. **QoS based selection:** The service selection process as a whole should provide a certain amount of flexibility to the requester, to allow QoS based selection.

In order to illustrate the problem statement let us consider the scenario as shown in Figure 2.3.

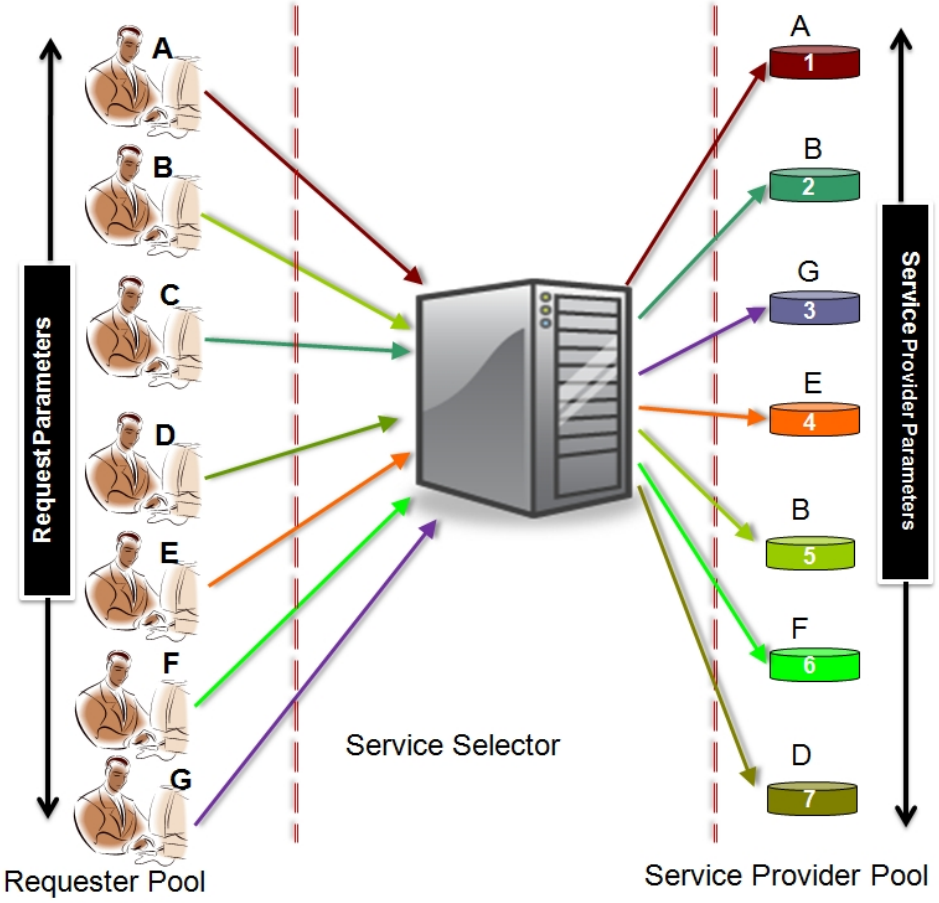

Figure 2.3: Selection process description

In this figure we can see that there are 7 requesters and services. Each requester has a set of specific criteria in term of QoS, which it seeks in a service provider. It provides those criteria within the request, which is primarily known as the QoS parameters. So typically requests from A will look like $[R_{id}, P_1, P_2, P_3, ..., P_n]$, where P_i is the QoS parameter, and R_{id} is the requester id. Similarly, the service providers advertise about their service provision capabilities. An entry in the service registry (which holds all the available service provider information) would typically look like $[S_{id}, Q_1, Q_2, Q_3, ..., Q_n]$, where S_{id} is the service id, and the Q_i is the QoS parameter that it is able to provide. An efficient service selection process, should be able to match the requesters and the services using these QoS parameters, within a reasonable time frame. The efficiency of the match is defined by the proximity of the QoS parameters of both the requesters, and the service providers. And, a good match is defined by a lower match score, indication a closer match. Moreover, the service selector should not only be able to handle large number of incoming requests, but should also be able to assign requesters competing for similar services efficiently, i.e., all the requester-service pairs should have a fairly balanced match score. Hence, balancing between the accuracy and the time efficiency of the algorithm plays a key role in these types of selection processes. Furthermore, the complexity of the problem increases with the increase in the number of (similar types of) requesters accessing the Grid services.

2.3 Objectives

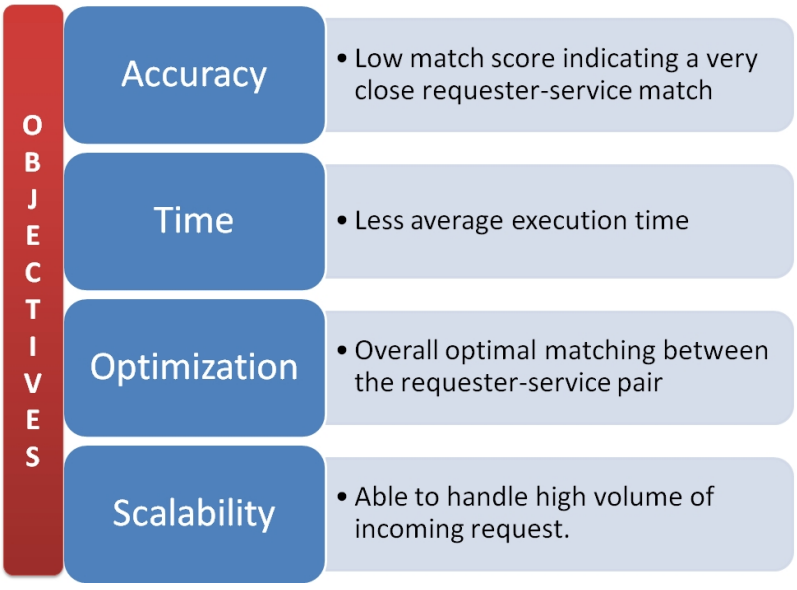

Figure 2.4: Proposed Approach Objectives.

In order to ensure an efficient selection, the above key objectives, as shown in Figure 2.4 is

required to be achieved. A brief summary of each of the objectives is given below:

- **Accuracy:** The accuracy of the match score should be high. It is one of the most important objectives of the service selection algorithm. As explained in the previous section, a good requester-service match depends on how closely the QoS parameters requested matches with the QoS parameters provided by the service providers.

- **Time:** Another key objective of an efficient service selection algorithm is to return the requester-service match score within a reasonable time. The problem at hand can be looked at as an assignment problem, where there are n number of tasks to be assigned to m number of agents. However, the standard assignment algorithm [18, 19, 20, 21] has a time complexity of $O(n^3)$. This makes the efficiency of the algorithm worse when more and more requesters required to be matched. Hence, a balance between the accuracy and execution time should be reached.

- **Scalability:** The accuracy of the service selection process should be independent of the large number of incoming requests, when the number of available services on the Grid is fixed. As the size of the Grid infrastructure can vary from regional, to across the nation, to continents, the size of the incoming requests can also vary from hundreds to thousands to a million requesters accessing the Grid at the same time. However, the time complexity of the requester-service matching process increases with the increase in the number of requesters. An efficient service selection algorithm should be able to handle the large number of incoming requests at the same time, by assigning them to an appropriate service provider within a reasonable time.

- **Optimization:** Finally, the overall match process must be as accurate as possible, i.e., all the requesters in the pool should be matched fairly, avoiding any kind of biased matches. This objective can be achieved by selecting the requester-service match combinations, with the highest sum of the individual requester-service pair match score. This objective, more specifically, avoids a requester to be matched to a service which provides services with attribute values higher than what is requested by the user, and vice-versa.

2.4 Summary

The chapter mainly focused on the motivation behind choosing this research topic. It emphasized on how the Grid is rapidly emerging as a SOA, and also stressed on how more and more industrial and research organizations, are depending on it to execute their computationally intensive application problem. The chapter also presented, that, the need for an efficient service selection process on the Grid becomes mandatory, as more and more organizations are seeking the Grid for a solution to

their computationally intensive problem. It also presented the problem definition of this research by focusing on what challenges are faced in designing an efficient service selection algorithm. The next chapter discusses the existing methods proposed to tackle these problems, and also presents the similarities and the differences between them.

Chapter 3

Related Work

Service computing's main goal is the "provisioning of services", such as services for high performance computing [22]. As the scope of high performance applications being deployed as services on the Grid becomes wider, the need for an efficient and sophisticated approach to deploy the service computing infrastructure becomes even more evident. Moreover, the increase in usage of wireless networks and mobile devices has made the computing environment more dynamic and sophisticated, as the resources, and services provided by them are entering and leaving the network at any given time. Resources are typically referred to the hardware and/or software that provides a certain kind of service. For example, a storage device is a resource that provides storage services. Likewise, a printer is a service, which provides a printing service. However, it is important to note that not all services are hardware resources. For example, a service may also be a software running on a server at some remote location [23]. In the Grid usually the monitoring and discovery systems [24], used to monitor and discover these services, is the information system component, i.e., it not only provides information about the available resources on the Grid, but also gives information about the status of the resource. Furthermore, the indexing services on the Grid collect the information about the available resources and publish them as the service providers property. The Grid users then need to select from these discovered services, an appropriate service to execute its job. The selection process becomes more challenging with the increase in the number of similar service providers (too many close matches), and also with the increase in the number of incoming requests accessing the Grid at the same time. Also, with the increase in the diversity of the services being deployed on the Grid, the need to allow the Grid users to discover these services appropriately and efficiently, becomes paramount. Taking all these facts into consideration, it is important to emphasize, that the service selection is a fundamental issue for the Grid, as it defines the process for locating service providers and retrieving service descriptions [25] to match it with the incoming requests.

3.1 Overview of Various Service Selection Methods

A fair amount of investigation has been done in the area of service selection on the Grid as mention in [26, 27, 28, 29, 30, 31]. And, many of the researchers have proposed service selection based on various

criteria such as, reputation, cost, time, while some are focused on an agent-based selection approach. A selection algorithm that is driven by a user supplied application deadline and a resource access budget is proposed in [32]. The algorithm selects services in such as way that user requirements are met (e.g. duration of execution), and yet it keeps the cost of computation at a minimum. The work in [33], provides a service selection process in which for each requester, a scheduler selects a set of servers that can handle the computation and ranks them based on the minimum completion time. Further, a context-aware service selection approach is mentioned in [23, 34, 35]. A context-aware service selection approach is able to use the context which refers to information about the surrounding environment, i.e., location of the user, location of the closely matched services, network conditions, etc, to match the requesters and the services. The applications uses this information to provide relevant information about the services to the user. Another approach, based on a multi-agent architecture and fuzzy techniques to represent the multi-granular capabilities of a web service, is mentioned in [36] and [37, 38]. The work presented in these, aims at improving the negotiation activity amongst various service providers through proactive evaluation of concept matching. The authors do this to solve semantic mismatches and to increase automated semantic interoperability. The authors uses an agent-based architecture to achieve service discovery. The agents return one or more services that can closely match the requests. The authors also exploit fuzzy multiset as a mathematical model to describe the multi-granular information included in the description of advertised services. From the above paragraphs its evident, that various scientists across the world have done some significant research on this topic, however not all approaches proposed in the literature is comparable to the approach presented in this thesis. A brief description of the four main categories of the service selection process, their similarities and differences with the proposed approach are given below:

- **Service Selection using the Classified Ads (ClassAd) mechanism:** The authors of [39, 40], present a general purpose resource selection framework. The framework uses a classified ads mechanism to discover and organize services based on the requester's choice.
- **Service Selection using the Semantic Matchmaking approach:** The general idea of this type of resource matching is to exploit the semantic web technology, and apply its advantages to the Grid. The scientists researching on this topic, provide matchmaking as an online service to solve the service matching problem on the Grid. Other variations of this concept are discussed below.
- **Service Selection based on QoS Parameters:** This category of service selection algorithm not only allows the service provider to advertise about their service providing capabilities, but also allows the requesters to state their requirements for the applications/services they need to execute. In this, the requesters play the most important role in the service selection process. The proposed approach is highly dependent on this feature, as it adds more flexibility on the user's part to choose a service from the list, based on a specific requirements.

- **Service Selection using Evolutionary Algorithms:** This category of service selection is fairly new and not a lot of research has been done in this area of service selection. In [1], the authors propose to use a variation of the Genetic Algorithm approach, to solve the service selection problem on the Grid.

3.1.1 Service Selection based on the Classified Ads Mechanism

Authors in [39, 41, 42, 43, 44, 45], presents a general-purpose service selection framework that supports both single-service and multiple-service selection on the Grid. The service selection process is based on the set-matching technique. Their work extends the Condor matchmaking framework, which is one of the foremost service selection frameworks in the history of Grid computing. The Condor Matchmaking [40] is based also on the semantic matchmaking procedure as mentioned above. In this the matchmaking is done based on symmetric, attribute-based matching, i.e., users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and finally informs the user upon completion. In this mechanism the requester submits jobs by using the following two methods:

1. If the user needs to submit the job indirectly to some service(s), it would do so using the ClassAd mechanism;

2. If the user needs to submit the job directly to a service, it would do so by submitting the job to a particular queue that is associated with the service.

The ClassAd mechanism used in the Condor matchmaking method, makes it a flexible and expressive framework for matching service requesters to service providers. It enables the requester to easily state its requirements (which is mainly execution time in these cases), and similarly the service providers can also specify their capacities and limitations (in terms of execution time) to execute a request, by this the authors have claimed to make the selection process more flexible, and user (requester) friendly.

3.1.2 Service Selection based on Semantic Matchmaking

One of the most prominent service selection methods as discussed in the literature including the semantic matchmaker [46, 47, 48, 25, 49], are explained in the following paragraphs. Essentially, the semantic matchmaker method proposes a solution for service discovery on the Grid, based on three selection stages which are context, semantic and registry selection. Instead of only performing service name matches which other common service discovery systems are restricted to, the semantic matchmaking framework provides a better service discovery process by using service semantics stored in ontologies. The framework permits Grid applications to specify the criteria with their

service request. These criteria are then matched with the services that are available, and hence enable interoperability in the matchmaking process [50, 25, 49].

A two step resource matching algorithm is also introduced in [51]. In the first step the authors classify the matches into four different levels. The levels are termed as: Exact, Plug-In, Subsume and Fail. Exact is the stage where all the parameters of service provider match exactly with the requirements of the requester, whereas Plug-In and Subsume are the stages where some of the parameters of the service providers do not match exactly with what is required by the requester, i.e., it is either a subset or superset of the parameters provided by the requesters. And finally, if none of the parameters of the service provider matches the requester parameters, the stage is called as fail. Then a ranking compare function is applied for the Plug-In and Subsume cases to produce a further detailed ranking. This function uses the semantic relation between the service requester and service provider (termed as resource requester and resource advertisements in [51]) to match them. The second step of the algorithm follows a time optimized policy. That is, once the first step of this algorithm returns the available resources, the resource which can complete the job in minimum time is selected. The semantic matchmaking process although a prominent service selection process, it has not been used in this research work, because the semantic based service selection is primarily useful when different types of services are being selected by the requester, which is not being considered in this research.

3.1.3 Service Selection based on QoS Metric

As described in [52], Quality of Service is an aggregated metric for describing characteristics of systems in areas, such as service oriented architecture, networks and distributed systems. The authors in [52] discuss that although there are QoS standards available for some domains such as Software Engineering, there is no mention about any standard QoS measures that should be taken into account for the Grid services. However, researchers in the field of web services (which is closely related to the Grid Services), propose various ways for grouping the QoS parameters into different categories such as:

- **Generic or domain specific QoS metric [53]:** Generic QoS measures includes features that are common for every service, e.g., price, execution time, reliability, etc. Whereas the second criteria refers to features specific to a particular domain, to which the application belong.
- **Deterministic and non-deterministic QoS metric [53]:** The deterministic group includes features that are known at the time when the service is invoked (such as cost of execution), whereas the non-deterministic group consists of criteria that are uncertain when the service is called (such as response time).
- **Objective and subjective QoS metric [54]:** Objective type QoS metric, includes features

such as availability, reliability, and response time, while the latter refers to the requester's experience in using the services, such as the QoS parameter reputation.

- **Run-time related QoS metric [52]:** This includes the features that can be evaluated dynamically, such as scalability, capacity, performance (execution time, latency, and throughput), availability, reliability, error rate, degree of correctness, and error handling.
- **Management and price related QoS metric [52]:** The management and price related QoS, consists of features which is related to standards, regulations, expected features/available features, and mainly the cost associated with the services.
- **Security related QoS metric:** Although very rarely stated as a QoS metric, this group includes the features which are access related such as, privacy, and data encryption.

There are different approaches described in the literature for selection of services that takes into account the QoS criteria discussed above. The authors in [55], proposed a multi agent-based architecture and the use of the semantic web in order to select the best service according to the consumer's preferences. The authors take into account, trust and reputation as the primary deciding factor while selecting the appropriate services. The authors in [53], consider these features in addition to other criteria such as execution price, duration, transactions support, compensation and penalty rate. The proposed framework in [53], evaluates the QoS of the available services by using the feedback and monitoring method. A QoS based service selection algorithm, as discussed in [1] is one that has provided the greatest inspiration for this work. It implements the service selection methods, based on five commonly used QoS criteria for services selection. The QoS parameters are given as follows:

- **Execution Price:** Is the cost incurred by a requester to use a service;
- **Execution Duration:** Is the time taken to execute a job, at a particular service provider site;
- **Reliability:** Is the probability that a request submitted by requester is correctly responded to within the maximum expected time;
- **Reputation:** Is the measure of trustworthiness of a given service (depends on requesters prior experiences of using the particular service);
- **Availability:** Is the probability that a service site is accessible by a requester, at a certain time.

The QoS metric used in this thesis, is similar to the ones stated above, as these are the most generic QoS parameters used in the service selection process. These QoS parameters, are also used to calculate the match score for a requester-service pair in the classical matchmaking algorithm, as proposed in [1]. The algorithm considers the first request from the list of requesters, scans the list of the services discovered, and assigns the service with the highest match score to the request. The selected service is then removed from further consideration. Similarly, the best matched service, for

the second requester from the list is determined based on the QoS parameters of the services that are available. Once a service has been found, which offers QoS parameters greater than or equal to what is requested, then the corresponding service is removed from the list of the services available. This goes on until each client request is matched with a service [1].

The algorithm can be split into two distinct parts: the first part calculates the match score, and the second part returns the best request-service match pair assignment. The calculation of the match score in turn consists of two parts - first the algorithm calculates the match value of each of the QoS parameters of the service requester pairs, and then it calculates the match score for the individual requester-service pair. The calculation of match score is done when it returns, a vector of match scores of the requester-service pair. In the second half of the algorithm, the algorithm checks for two conditions. Firstly, it checks if the current match score of a requester-service pair is greater than the match score returned, when the same request was matched the previous service and secondly, it checks whether the service is available. If both conditions are true the requester is assigned to that service.

The two primary advantages of this approach are:

1. Each service requester is considered only once and hence, it avoids redundancy in the service allocation process;
2. Execution time is fairly reasonable.

3.1.4 Service Selection Based on Evolutionary Computation

As a solution to the problems faced in the classical matchmaking the authors of [1] have proposed to use the genetic algorithm approach. This research work has introduced evolutionary algorithms, more specifically genetic algorithms to the field of service computing. A genetic algorithm (GA) is a heuristic used to find approximate solutions to problems which do not have a single best solution. GA applies the principles of evolutionary biology to computer science, and the steps in the algorithm takes inspiration from "Darwins theory of evolution" [56]. GAs use human evolutionary techniques such as inheritance, mutation, natural selection, and crossover [57]. A detailed description of the GA, its variations and other evolutionary techniques is given in the following chapter. GAs are typically implemented as a computer simulation in which a population of solutions to a problem evolves in each generation to find a better solution, i.e., the best requester-service match combination for this problem. The solutions are represented as a chromosome which can undergo genetic modification in each generation. In [1], the chromosomes are a randomly generated permutation of integers, where each integer represents a service requester. The length of the chromosome therefore depends on the number of requester. The authors of [1], have considered the services as a static integer permutation from 1 to N. There are many variations of GAs in literature, and the one used by the authors of [1] is the Non-Dominated sorting genetic algorithm-II (NSGA-II) [58]. The

NSGA-II did show remarkable results, with a large number of requesters being matched efficiently with the number of available services, and the service selection algorithm did return high match scores.

3.2 Comparison of the Related Work

Using the condor and set extended matchmaking procedure, it can be seen that the service selection process is a part of the whole procedure, which includes two phases. Phase 1, includes discovering the available service, and phase 2 includes selecting the best match service among them. The Condor matchmaking method stresses more on how to effectively schedule a job on the whole, whereas the method presented in this research focuses mostly on the service selection procedures to be used while matching a particular service to a request. Moreover, the authors of the condor-matchmaking and set-extended matchmaking quantifies the performance of their approach, based on the time required to execute the job, and therefore takes into consideration only one parameter, i.e., the execution time the jobs. Whereas, in this research five most generic QoS parameters are considered to match each of the service requesters and the service providers. The multiple parameters considered provide more flexibility, however it also adds more complexity to the selection process. Hence, the matchmaking process requires advanced optimization techniques, especially when large numbers of requesters are competing for similar kind of services at the same time. Similarly, the authors of [47, 51] have also given importance on the *time optimized policy*. Whereas, the research work presented here takes into consideration a novel concept of a *user parameter selection policy*, whereby the user has the flexibility to emphasize on the QoS parameter that holds a priority. For example, in a scenario where a group of companies located in different cities, meet virtually to reach a decision of opening a new factory at a particular location. During the decision making process, different types of applications are been executed simultaneously from different locations on the Grid. In this scenario, the users of the Grid might want to speed up the job execution process and hence time plays an important role. On the other hand, there might be a case where a small scale company decides to use the Grid services and for them the cost of execution might be more of a priority than the execution time, i.e., they are willing to compromise the time to obtain a lower cost. Our research has considered scenarios similar to this, to make the infrastructure more flexible. The classical matchmaking algorithm, based on a QoS metric mentioned in [1] performs well when there are less requesters as compared to the number of services available. This is because in this case the requesters has more services to choose from. However the problem arises when there are equal number of services and requesters, and most of the requesters are competing for similar services. The disadvantage of this lies in the fact that the requesters later in the list are very likely to get assigned to a service with a worse match score. And the likelihood of the request being assigned to poor service providers increases with the increase in the number of requesters accessing similar kind of services.

The algorithm's performance also depends on the position of the requesters and the services in their respective list, because the algorithm matches the requesters to the services sequentially in accordance to their appearance in the list. This particularly results in a bad match score if high demanding requesters, with high QoS parameter values appear later in the list. Finally, in [1] although the results achieved using a GA as the service selection method were quite encouraging, it was achieved at the cost of longer execution times, e.g., from the experimental results it can be seen that for 200 requesters and 200 services, the GA was expected to outperform classical matchmaking in 172s. With the increase in demand of the service computing infrastructure, and with more organizations and/or individuals depending on them, the time within which appropriate services are selected plays a very important role. Hence, it is crucial to find a selection algorithm which balances the accuracy of the match score and the latency, at the same time.

3.3 Summary

Some have said that service computing will become a utility just like any other utility in the near future. A service-rich environment in the future, where more and more industries, researchers, and other organizations are accessing the Grid to execute their requests can clearly be envisioned. Hence, the need for an efficient service selection algorithm to match the above criteria has become paramount. To facilitate this scenario, the Grid service selection algorithm should be able to tackle a large amount of incoming requests, assign them to appropriate services efficiently, and most importantly match the requests to the services in a way which avoids any kind of biased matching, achieving an overall higher matchscore accuracy. To achieve this goal initially an enhanced version of the classical service selection algorithm based on the constraint satisfaction problem solving approach is proposed in this research. In this algorithm the requesters are matched with a service after considering the entire list of service providers, instead of considering them sequentially. The problem of assigning the requesters appearing later in the list to the inefficient services, was solved considerably by changing the method of choosing the best matched service from the service provider list. The research focuses on making the service selection procedure more efficient. Hence, to produce better results under the circumstances where large numbers of requesters are competing for similar services within a reasonable time, a swarm intelligence based strategy is used. Finally, as the problem of matching requester-service pairs can essentially be considered as an assignment problem, where a set of n tasks is assigned to a set of m agents. The proposed approaches are compared to a standard assignment algorithm - Munkres Algorithm - to compare for their accuracy. This comparison gives a fair idea about the efficiency of the proposed algorithms. In the next chapters the proposed service selection algorithms are introduced for the selection of Grid services, which overcomes most of the setbacks seen in the other algorithms mentioned in the above paragraphs.

Chapter 4

Proposed Service Selection Algorithms for the Grid

The Grid paradigm enables the coordinated sharing of a large number of geographically dispersed heterogeneous resources amongst different user communities. It is based on the idea of resource abstraction and user abstraction that together defines a virtualization layer known as VO, over the actual resources. Within this layer, resources can be dynamically grouped into pools and, can be accessed by users with appropriate credentials. When the users express their needs over a particular set of resources, the virtual resources are mapped to the actual resources using mapping functionalities. And once the list of available resources is found, a service selection algorithm is used to select an appropriate resource from the pool [59]. Two service selection algorithms, based on the QoS based selection method, are introduced in this chapter. The content of the chapter can be split into the following aspects:

- **Issues addressed:** This section focuses on the issues addressed while designing the proposed service selection algorithms.
- **Key concepts inspiring the proposed service selection algorithm:** The section discusses the key concepts on which the proposed service selection are based, i.e., a) the Quality of Service metric, and b) Evolutionary Algorithms.
- **System architecture:** This section explains the service selector architecture implemented. It briefly describes all the components of the architecture, and also discusses how the proposed algorithms are integrated to it.
- **Proposed selection algorithms:** The section focuses on the service selection algorithms that are being proposed in this thesis to address the issues mentioned earlier.
- **Assignment Problem algorithms:** The problem of allocating a request to an available service on the Grid can be seen as a classic assignment algorithm. Hence, the proposed algorithms are compared to a traditional assignment problem algorithm named, Munkres Algorithm. A brief overview of the assignment problem algorithm is presented is this section.

4.1 Issues Addressed

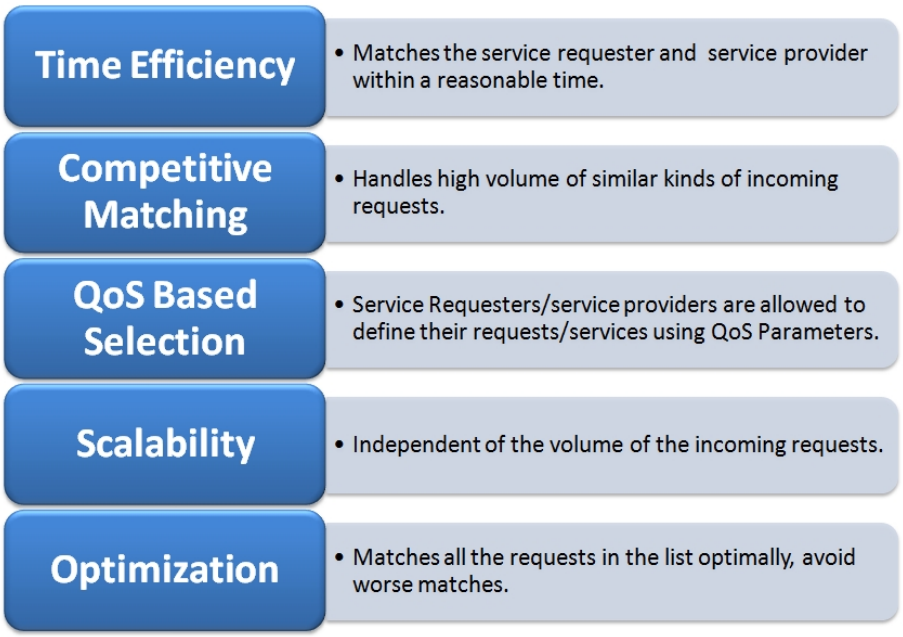

Figure 4.1: Issues Addressed.

The background research of the existing service selection methods, shows that the requesters usually are assigned to the available services sequentially, and this method favors the requesters appearing earlier in the list. It is also seen that the service selection criteria is solely based on the minimum execution time of the service providers. Although, the execution time is one of the most important selection criteria, it cannot be considered as the only benchmark to select services. These vulnerabilities of the existing algorithms were considered while designing the proposed service selection algorithms. Hence, multiple QoS service selection concepts are introduced, based on which the available services are selected from the Grid. The issues addressed while designing the proposed algorithms are briefly listed in Figure 4.1 and are also summarized as follows:

1. Assigning a large number of client requests to the available services simultaneously, and efficiently. The time the algorithm takes to match a set of requesters to the available service is crucial. Hence, the latency of the algorithm is one of the most important issues addressed in the proposed approach;
2. The proposed algorithm should be able to handle scenarios where a large number of requesters have requested for similar kinds of services, defined by a set of QoS parameters. This would result in a competitive matching scenario. An efficient service selection algorithm should be able to match the requests for similar services in a more accurate way without compromising the overall match score;

3. The next issue addressed is to provide the requesters with flexibility to have multiple service selection criteria based on a set of QoS parameters. This overcomes the shortcomings of the existing service selection algorithms, where execution time was the only service selection criteria. Similarly, the service providers can also advertise their limitations and capabilities to execute a request by defining a set of QoS parameters;
4. The algorithm should be scalable, i.e., the performance of the algorithm should remain unchanged or only change minimally even when there is a large number of incoming requests, for a fixed number of available services on the Grid. In this case the algorithm should be able to select, and match the requesters resulting in an accurate match;
5. Guaranteeing *optimized* assignment of *all* the requests to the available services is also one of the key feature of the proposed algorithm. The proposed algorithm, avoids matching the requests later in the list with a worse match score, by considering all the incoming requests at the same time, instead of assigning them sequentially in the list;

The aim of this research work is to investigate and design an algorithm that addresses the above mentioned issues, by enabling optimized selection of services in the Grid, and at the same time allowing the requester to express their requirements and preferences regarding the services to be selected. The algorithm finds a suitable solution i.e., an appropriately matched service based on the selection criteria provided by the user when submitting the request to the Grid. Furthermore, these algorithms can be integrated with the Grid scheduling process in order to improve the selection capabilities of the system. This thesis primarily introduces the concept of a swarm intelligence based service selection methods to the Grid. It also propose an enhancement of the classical service selection algorithm. Both of these algorithms are designed to achieve the above goals. Finally, a set of experiments and evaluations are performed, to test the efficiency of these algorithms.

4.2 Main Concepts

The problem of service selection on the Grid consists of finding an efficient algorithm which can match multiple client requests and the service providers efficiently, while optimizing the overall match score at the same time. Hence, the proposed approaches are primarily based on the following concepts:

- **QoS Metric:** This helps the requesters to specify preferences of the services; hence provides more flexibility to the user of the Grid. Additionally, a QoS metric is also considered in terms of the services to express their limitations while executing the requests.
- **Evolutionary Algorithm:** The evolutionary algorithms, also known as EAs are briefly introduced in this section. There are various type of EAs in the literature. Only two types, namely Genetic Algorithm, and Particle Swarm Optimization Algorithm are explained and

used in this research. The strength of these algorithms lies in the fact that, they usually reaches the solution more quickly than the traditional algorithms. The reason for this being is because they apply heuristics to reach the final solution. It also important to note that these algorithms come with a trade off between accuracy and the time efficiency of the algorithm. Hence, the EA strategy is used in this research to optimize the complex matchmaking process.

A brief discussion of both concepts are given in the following sections, followed by the description of the two proposed approaches.

4.2.1 Quality of Service Metric

Figure 4.2: Quality of Service parameters considered in this research. [1]

The QoS metric plays an important role in the service selection process on the Grid. It is especially helpful when there is more than one similar service provider available for a particular client request. In this situation, a set of QoS parameters helps to allocate the request to the most qualified service. These QoS parameters can be adopted by all Grid services, for example, for their pricing, execution time and availability. However, when dealing with dynamic components such as Grid Services, it is hard to observe all of their possible features. Hence, this research has focused on the most commonly used QoS criteria as listed in Figure 4.2. They are particularly categorized as most generic QoS parameters, because they can be applied to any service. Also, these services are able to define the requester/service attributes completely. Availability, cost of execution, and the response time are the most popular ones, as they provide a clear overview of quality of the service, and at the same time they can be evaluated easily. This research also focuses on two other

QoS parameters, namely reliability and reputation. The proposed algorithm is quite flexible, hence more QoS criteria (both domain and non-domain based) can be added without altering the core steps of the algorithm. The QoS criteria used in this research are briefly explained in the following paragraphs.

Execution Price

The execution price, also referred as the *cost*, of service S, is the cost incurred (in terms of \$) by the service requesters to utilize the service in order to compute an operation O, at any given time t. The service providers advertise the execution price of each operation in the registry. Whenever a requester wants to access a particular type of service, it would look up the registry to see at what price a particular service provider is offering a service. The metric can be formally represented as, $Q_{price}(S, O)$, where, S is the service to be used to execute the operation O.

Execution Duration

Each service provider advertises their average execution duration for a particular type of service. The average execution duration, $Q_{exec}(S, O)$ reflects the difference between the time when the execution of operation O starts and when the execution of the operation ends, while using service S. The difference between the start and the end time of an execution, essentially includes the processing time $T_{procs}(S, O)$ and the average transmission time $T_{trans}(S, O)$. The average transmission time, $T_{trans}(S, O)$ is calculated by taking the average of the past transmission times using the following formula, $T_{trans}(S, O) = \frac{\sum_{i=1}^{n} T_i(S,O)}{n}$, where $T_i(S, O)$ is a past observation of the transmission time usually stored in the log file, and n is the number of execution times observed. Considering that the execution duration $T_{exec}(S, O)$ is the sum of the processing and the average transmission time of the operation O on service S, operation O while using service S is $Q_{exec}(S, O) = T_{trans}(S, O) + T_{procs}(S, O)$.

Reliability

The reliability, $Q_{rel}(S)$ of a service S is the probability that a request is correctly responded to within the maximum expected time duration. This is important especially when the requests are being executed remotely. This helps to invoke the request execution for the second time if the first request fails due to network failure or overloading. The reliability of the service depends on the system configuration of service provider and also the network connections between the service requesters and providers. Reliability is usually computed by logging the information of all the past executions of a particular service. The probability of a service being more reliable increases with the increase in the number of successful executions. It is calculated by dividing all the occurrences of the successful execution of services by the total number of executions that the service S has run. It can be represented formally as, $Q_{rel}(S) = \frac{N_{Suc}(S)}{N_{total}(S)}$, where $N_{Suc}(S)$ is the number of times

service S has successfully executed the request within the maximum expected time frame say t', and $N_{total}(S)$ is the total number of execution service S has executed.

Availability

The availability, $Q_{avail}(S)$ of a service S is the probability that the service is accessible by an end user. The value of the availability of a service S is computed by taking into consideration the time t (in milliseconds) in which service S was available over that time. This period is a constant decided by the service providers. The value $Q_{avail}(S)$ can be formally represented as $Q_{avail}(S) = \frac{T_{avail}(S)}{\Theta}$, where $T_{avail}(S)$ is the total amount of time in which service S was available during the last Θ seconds. This information is usually advertised by the service provider and is kept as a log in the service registry.

Reputation

Reputation $Q_{rep}(S)$ of a service S, is a measure of its trustworthiness and it depends on end user's experiences of using service S over a certain period of time. As this value depends on the end users perception about a particular service provider, different end users may have different opinions about the same service. Hence, the value of the reputation is computed by taking the average ranking given to the services by end users. It is also important to note that in this case, end users are usually given a range to rank the service providers. The value of the reputation for a service $Q_{rep}(S)$, can be formally represented as, $Q_{rep}(S) = \frac{\sum_{i=1}^{n} R_i}{n}$, where R_i is the end user's ranking on a service's reputation, n is the number of times the service has been ranked.

4.2.2 Evolutionary Algorithms

Evolutionary Algorithms are a set of algorithms that take inspiration from natural phenomena of human evolution, first introduced by Darwin [60, 61]. The algorithm primarily includes initialization of population of candidate solutions, and then performing a series of steps which includes reproduction, mutation, recombination, and selection on them iteratively, to find the better solution (fittest individual) to the problem within the problem space. Various types of Evolutionary Algorithms as listed in Figure 4.3 can be found in literature [62, 63, 64]. The most important ones are discussed in this research, namely: the genetic algorithm and the particle swarm optimization technique. A brief description of both is given in the following sections.

Genetic Algorithms (GAs)

The GA concept is highly derived from the notion of human evolution. The GA follows an iterative sequence of steps; it starts with a population of randomly generated individuals. In each generation of evolution, the fitness of individuals from the pool is evaluated. The fittest individuals, in terms

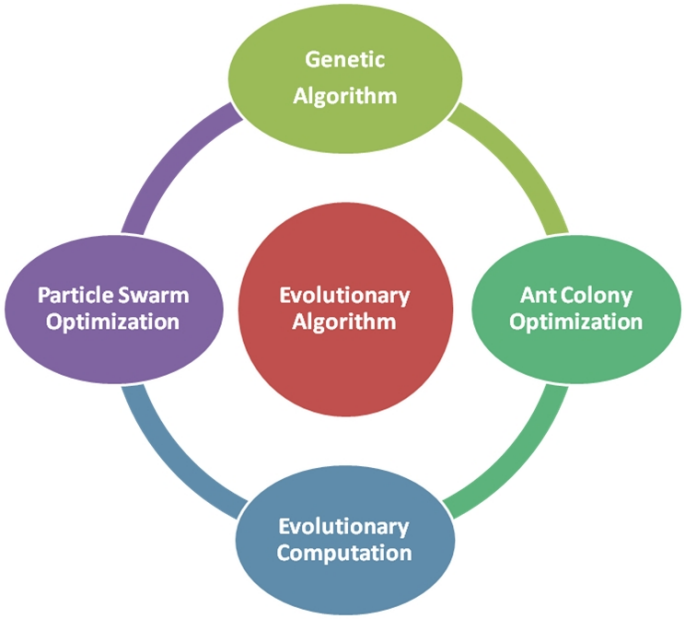

Figure 4.3: Evolutionary Algorithms.

of the best match score are found at each generation. In the next step two fittest individuals are randomly selected from the pool, and then they undergo crossover and mutation operation to form the new generation for the next round of evaluations. A brief description of the algorithm can clearly be seen in the Figure 4.4 [65].

These mutation and/or crossover steps are repeated until the maximum number of iterations is achieved. In the next generation the same steps are followed and the individuals are evaluated; the fittest individuals are selected and the process repeats until either a predefined number of iterations are completed, or the fittest individual is found. The Pseudo code for the genetic algorithm is given Algorithm 1.

The pseudo-code shows the steps followed in a genetic algorithm. Although, this is an example of the classical genetic algorithm, there exist several well accepted variations of genetic algorithms depending on the representation of the individuals, how each step of the algorithm or the fitness function is designed. One such variation is the Multi-Objective Optimization Genetic Algorithm (MOO-GA). The idea behind these types of genetic algorithms is that instead of optimizing one parameter at a certain time, multiple parameters are optimized simultaneously. The multiple objectives to be optimized are usually incorporated in the fitness function, and the complexity of the algorithm increases with the increase in the number of objectives. There are various algorithms based on this concept, such as Non-dominated sorting GA-II (NSGA-II) [58, 66], which are one of the most prominent multi-objective optimization techniques. Some of the others being: Vector Evaluated Genetic Algorithm (VEGA) [67], and Niched Pareto Genetic Algorithm (NPGA) [68].

Algorithm 1: GA Pseudo Code

```
begin
    g ⟵ 0 (Generation counter)
    for P[i] in P[n], i < n do
        Initialize
        evaluate
    while !maxIterations do
        g ⟵ g + 1
        select P(g) from (g − 1)
        crossover P(g)
        mutation P(g)
        evaluate P(g)
end
```

Particle Swarm Optimization

Particle swarm optimization (PSO) [69, 70], is a population based stochastic optimization technique developed by Eberhart and Kennedy in 1995. It was inspired by the social behavior of bird flocking or fish schooling. PSO shares many similarities with GA as in this, the system is initialized with a population of random solutions and searches for a global optima by updating generations. However, unlike GAs classical PSO has no evolution operators such as crossover and mutation. In PSO, the particles (which are the potential solutions), migrate through the problem space by following the best global guide (fittest particle). The goal is to find a better solution of an objective function defining the problem. The objective function can either be a minimizing or maximizing depending on the problem. In this research the problem is a maximization problem where the proposed particle swarm algorithm seeks to maximize the requester-service match score.

Figure 4.5 shows the flowchart of the PSO algorithm [71]. As described in [70, 72], each particle keeps track of its coordinates in the problem space which are associated with the best solution (similar to the concept of fittest solution in GA) it has achieved so far. This global best value is also stored by the particles for each iteration (equivalent to a generation in a GA), and this value is called p_{best}. The second "best" value, that is tracked by the particle swarm optimizer, is the best value obtained so far by any particle in the swarm. When a particle takes the whole population as its topological neighbors, the best value is a global best and is called g_{best}. The particle swarm optimization concept consists of, at each time step, changing the direction of each particle's search toward its p_{best} and g_{best} locations. Acceleration is weighted by a random value, with separate random numbers being generated for acceleration toward p_{best} and g_{best} locations. A brief description of the above algorithm is given in Algorithm 2.

In the past several years, PSO has been successfully applied in many research and application areas, such as in [73, 74, 75, 76]. Application of PSO in a Grid system is however a novel approach.

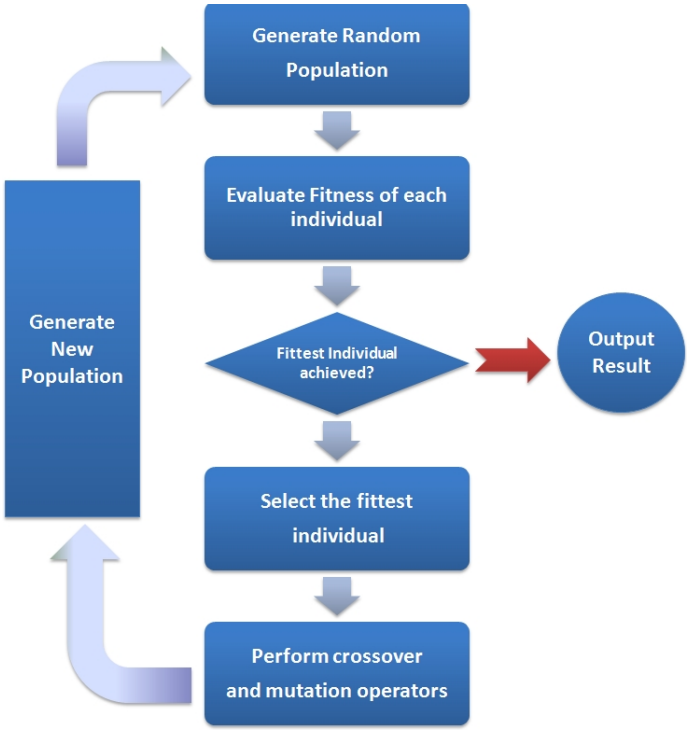

Figure 4.4: Genetic algorithm flowchart.

The classical PSO algorithm fails to tackle problems where multiple objectives needs to be optimized. Hence, this thesis demonstrates the use of an advanced particle optimization technique to design the service selection process on the Grid. A brief introduction of the advanced PSO version used in this research is given in the next section.

Multi-Objective Optimization Using Evolutionary Algorithms

Initially, the PSO algorithm had only been applied to single objective problems. However, later it was found that there are a large number of applications where a number of competing quantitative measures defines the quality of a solution such as the multiple QoS parameters. For example, while selecting the Grid services, the requester may wish to give the availability QoS parameter a less leverage, while giving more priority to the cost and time QoS parameters of the task execution. These objectives cannot be met by a single objective problem solving approach, so, by adjusting various QoS parameters, the requester may need to discover what possible combinations of these objectives are available, given a set of constraints. The goal of multi-objective algorithms (MOAs) [77, 78] is to locate the solutions that can be qualified as the best solution [79, 80]. In the literature it is already found that the Multi-Objective Evolutionary algorithm (MOEA) [79, 81, 82] has done a great job of finding a suitable global optima for these kind of problems. And hence, the idea of introducing MOPSO for selecting Grid services, did not seem particularly impractical. There are a

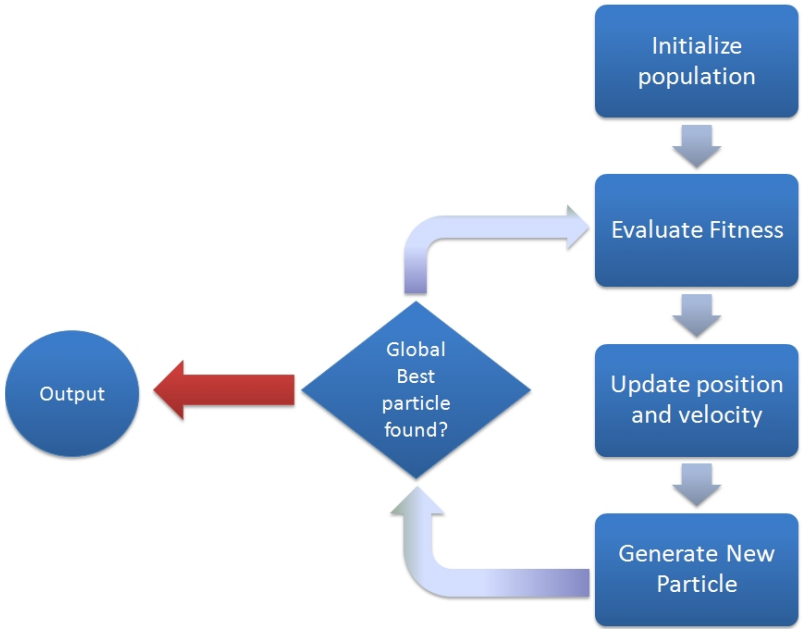

Figure 4.5: Particle swarm optimization flowchart.

number of different studies published on multi-objective PSO (MOPSO) such as in [83, 84, 85, 86]. Although the MOPSO is an extension of MOEAs for the PSO, the PSO heuristic puts a number of constraints on MOPSO that MOEAs are not subject to. As mentioned in [87], in PSO the swarm population is fixed in size, and its members cannot be replaced, but only can be adjusted by their p_{best} and the g_{best}, that are easy to define. These characteristic fits the problem structure in this thesis, as in the case of services, the number of services available at an instance of time is fixed, and also the services or requesters cannot be replaced from the pool. The MOPSO used in this research finds a better solution by using the concept of crowding distance technique. The algorithm including this concept is explained shortly in the following sections.

4.3 Service Selection Architecture

A general-purpose service selection framework based on the QoS based matching technique is introduced. A brief description of the architecture is given in Figure 4.6. In this, the service selector accepts a set of requests and finds a set of services with the closest match scores based on the QoS metrics information provided by requesters. It also provides an additional feature for the users to specify the preference value range used to prioritize the QoS parameters, which makes the algorithm more user friendly. The service selector has the following components:

- **Requester Pool:** It is the pool of incoming requesters. The incoming requests are queued in this pool, until a threshold number of requests has been reached. Once the required number

Algorithm 2: PSO Pseudo code

```
begin
    for P[i] in P[n], i < n do
        Initialize
    while !maxIterations do
        for P[i], where i = 1 to n do
            Calculate fitness value
            if fitnessValue > p_best then
                p_best ⟵ fitnessValue
        for P[i], where i = 1 to n do
            Find in the particle neighborhood, the particle with the best fitness
            Calculate new velocity
            Update particle position
end
```

of requesters are in the pool, the requests are sent to the request-service matcher, which in turn matches these requests to the available Grid services. The information sent to the request-service matcher, primarily includes the request ID and the QoS parameters provided by each requester and the range of preference values provided by the them.

- **Service Pool:** It is the pool of various types of services available on the Grid. Each service in this pool, has an unique ID, and also has a specific value for the QoS metric.
- **Request-Service Matcher:** The request-service matcher is the main component of the architecture. It includes the proposed service selection algorithms to match the incoming requests with the set of available services. It takes the user request information and the service information from the incoming requests and the service registry respectively. After acquiring the list of incoming requests and the available services, the request-service matcher executes the algorithm to generate the match scores.
- **Service Monitor:** The service monitor acts as a simulated Grid Index Service (GRIS) [88]. It is responsible for querying the Monitoring and Discovering Services [88, 89] of the Grid, to discover and access the configuration and status information of the services, such as their availability, CPU load, execution time etc.
- **Service Registry:** The information about the services, after being monitored by the service monitor is sent to the service registry. The service information stored in the registry primarily includes, the service ID, the URL to locate the service, a brief description of the service, and the QoS parameters of the service. This QoS information such as its availability, cost, execution time, etc., is updated in a regular time interval, to include the most recent attributes of a service. The request-service matcher uses the QoS metric stored in this registry to match

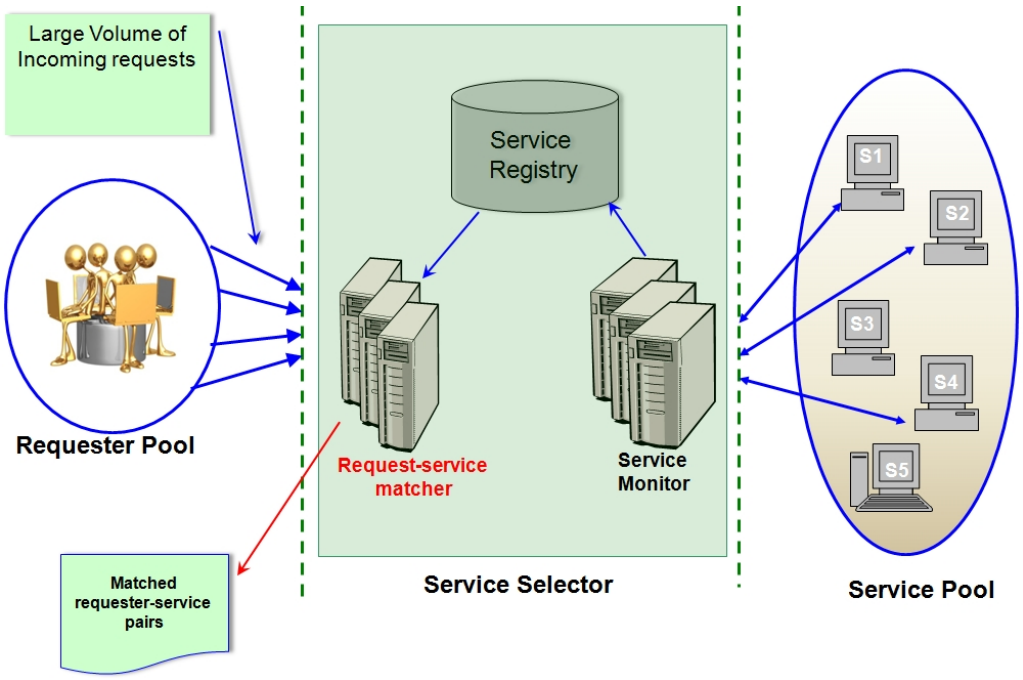

Figure 4.6: Service selection architecture.

a incoming request to an appropriate service from the pool.

4.4 Proposed Service Selection Algorithms

In this section, the service selection algorithms based on constrained satisfaction and Particle Swarm Optimization techniques are presented. Two new service selection algorithms are introduced: the Constraint-Satisfaction based Grid Service Selection algorithm - *CS Selection Algorithm* (CSS Algorithm) and the Particle Swarm Intelligent based Grid Service Selection algorithm - *PSI Selection Algorithm*(pronounced as SCI). The former algorithm is an enhancement of the classical requester-service selection algorithm, whereas the later is a novel approach based on multiple objective particle swarm optimization algorithm, which uses crowding distance in the service selection process. The motivations for researching and designing these algorithms was primarily based on three simple goals. Figure 4.7 briefly explains those, however a detailed explanation is given below:

- **Goal 1:** Our first step was to enhance the existing traditional selection algorithm. This was done by modifying the existing algorithm to include the Constraint Satisfaction based optimization approach in the service selection process. The aim was to consider all the incoming requests simultaneously as a set to be matched with the available services, in order to avoid matching the requests later in the list with a worse match score.

- **Goal 2:** Our second step was to introduce artificial intelligence based techniques, more

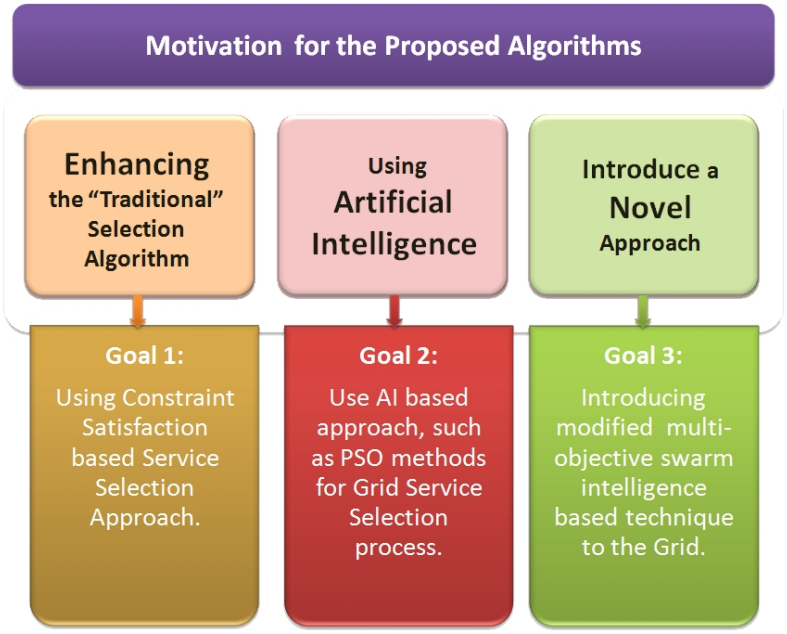

Figure 4.7: Motivation for Selecting the proposed algorithm.

specifically the concept of *Swarm Intelligence* to the Grid service selection process. This thought was inspired by the fact that, the task of service selection requires an efficient and accurate selection method because:

1. Firstly, the service selection process on the Grid requires more than one objective to be considered while assigning the request to an appropriate service. The objectives are usually defined by the QoS parameters. In this case the QoS parameters are: execution time, cost, availability, reliability and reputation. Also, the matching process becomes more challenging if the objectives have varying preference values provided by the requester;
2. Secondly, the selection process is a polynomial time problem. The traditional algorithms find a solution in polynomial time. In this case heuristics based methods help to achieve a balance between accuracy and time efficiency of the algorithm.

- **Goal 3:** Our third step was to introduce a novel multi-objective PSO based algorithm for the selection process. The swarm intelligence based method has never been used in the service selection area for the Grid. So the aim was to explore the possibilities of swarm intelligence, in this area of the Grid.

In the next section, the CSS algorithm in the context of an enhanced version of the traditional requester-service matching algorithm is presented. Then the PSI algorithm is introduced, and discussed in the next section as a further improvement to the CSS algorithm.

4.4.1 CSS: The Constraint Satisfaction based Selection Algorithm

In this research we have assumed that each service and requester, have five QoS parameters, such as service S_i and requester R_j have the following QoS metric $[S_{i1}, S_{i2}, ..., S_{i5}]$ and $[R_{j1}, R_{j2}, ..., R_{j5}]$ respectively. The QoS parameters $[R_{j1}, ..., R_{j5}]$ specified by the requesters, is defined as $R_{jx} = w_x * A_{jx}$. In this, R_{jx} is the value of the x^{th} QoS parameter, and the w_x is the weight value provided by the requester for the x^{th} parameter. For example, if the attribute "availability" is an important service selection criteria to the user then it would set it a high weight value between the range of 0 and 1, and vice versa. This feature gives the requesters the flexibility to define their preference regarding a QoS parameter. Finally, A_{j1} is the actual QoS parameter value that the user seeks in an appropriate service. It is important to note that the values of these QoS attributes should be between the of range 0 to 1. The five attributes of each service and requester are recorded in two 2-dimensional matrices. The dimensions of these matrices are $n \times 5$, and $m \times 5$, where n is the number of requesters and m is the number of services. The first step of this algorithm is to calculate the match values for each QoS parameter of all possible requester-service pair. The match value represents the distance between QoS values of the requester-service pairs. The match value $MV_{(R_{jx},S_{ix})}$ for the x^{th} QoS metric for R_j and S_i requester-service pair is calculated as follows:

$$MV_{(R_{jx},S_{ix})} = 1 - (\frac{|R_{jx} - S_{ix}|}{R_{jx}}) \qquad 4.1$$

Whereby the value of i and j, ranges from 1 to n and m respectively, where n and m are the number of requesters and services at a certain time; x represents the QoS parameter; R_{jx} represents the service requester R_j's value for the x^{th} QoS parameter; S_{ix} represents the service provider S_i's value for the x^{th} QoS parameter. The match value is subtracted from 1, which means, the higher the value of $MV_{(R_{jx},S_{ix})}$ the better is the match. The overall match score $MS_{(R_j,S_i)}$ for the requester-service pair R_j and S_i is calculated as follows:

$$MS_{(R_j,S_i)} = \frac{\sum_{x=1}^{Q} MV_{(R_{jx},S_{ix})}}{Q} \qquad 4.2$$

Whereby the value of Q is the number of QoS parameter considered for the problem. In this case the value of Q is set to five. The CSS algorithm calculates the match score for each requester-provider pair using equations 4.1 and 4.2. It stores the match score for each requester-service pair in a MS matrix as follows:

$$\mathbf{MS} = \begin{pmatrix} MS_{11} & MS_{12} & . & . & . & MS_{1m} \\ MS_{21} & MS_{22} & . & . & . & MS_{2m} \\ .. & .. & .. & .. & .. & .. \\ \\ MS_{n1} & MS_{n2} & . & . & . & MS_{nm} \end{pmatrix} \qquad 4.3$$

Whereby the value of MS_{nm} in the above matrix is the match score of the n^{th} requester and the m^{th} service. Once the matrix is formed, the highest value from each row is selected. This value

represents the best possible match of the corresponding requester and service. In the event, where more than one requester is assigned to a single service, the constraint satisfaction based problem solving approach is applied.

The constraint in this problem is to assign a single requester to a service, without affecting the overall match score of the requester-service pair significantly. This constraint is based on the assumption that a service can only serve one requester at a certain time. And hence, in order to satisfy this constraint the second highest match score for the service-requester pair is considered. This is done in the event if the service with the maximum match score is already assigned to a different requester. Also, in this scenario the requester with the minimum difference between the highest and the second highest match score is assigned to the second highest match score. For example, if R_1 and R_5 both are assigned to S_7, the second highest match scores for both R_1 and R_5 will be taken into consideration. The second highest match score with the least difference between the highest match score and the second highest match score will be chosen. This process continues until each service is uniquely matched with a single requester. The strength of the algorithm lies in the fact that it matches the requesters with available services in a fairly optimized manner avoiding the problem of poor matching of the requesters appearing later in the list, as seen in the classical matchmaking algorithm. The actual source code of the algorithm can be found in Appendix A, Section A.3.

4.4.2 PSI Selection Algorithm: The Particle Swarm Intelligence based Selection Algorithm

This section focuses on the proposed Particle Swarm Intelligence based Service Selection algorithm for the Grid services, based on an advanced multi-objective PSO algorithm. The approach relies on the concept of crowding distance, which helps to get a better solution during the match process. A brief description of this concept is given in the next few paragraphs, following which the actual description of the algorithm is provided. The crowding distance concept helps to choose the global best solution in the swarm [58]. It is calculated by first sorting the set of solutions in descending order of the personal best solution in the swarm. And, then the global best is selected from a predefined percentage of this sorted list. For example, if the problem is a maximizing problem, the global best is selected from the top 10% to 20% of the solution, while if the problem is a minimization problem the global best is selected from the bottom of the sorted list. [90]. The motivation for using the crowding distance based PSO algorithm amongst all other multi-objective optimization technique lies in the fact that it provides a better method to select the global best from the personal best solution set achieved in each generations. The crowding distance concept not only helps to diversify the search for the global best solution, which provides a better way of selecting the global best guide, but also avoids getting stuck into the local optima in the solution space. This

overcomes the problem of "biased" matching, found in the traditional requester-service matching algorithm, more efficiently. The traditional requester-service matching algorithm fails to assign services with the requesters appearing later in the list with a good match score. This is because the matching process is sequential, and not simultaneous. Hence, the requesters appearing earlier in the list dominates the overall match score. Using the crowding distance based PSO algorithm, helps to optimize the service requests efficiently by returning a better solution to the problem. Thereby, guaranteeing an unbiased matched solution by using this method. The multi-objective particle swarm optimization algorithm using crowding distance, introduced by Raquel in [90], has provided a great inspiration for this research. The algorithm provided in [90] has been modified to fit the Grid service selection problem description. The PSO algorithm used for solving this problem is named PSI Selection Algorithm, and can be explained as follows:

1. **The Objective Function Definition:** The definition of objective function is one of the most crucial steps in a swarm intelligence based algorithm. The objective function for this problem is based on the match score Equation 4.3, mentioned earlier in this chapter. It is mathematically represented as $f(MV_{(R_{jx},S_{ix})}, N, n)$, where $MS_{(R_j,S_i)}$ is the average of the sum of match score of each requester-service pair, such as:

$$MS_{(R_j,S_i)} = \frac{\sum_{i,j=1}^{N}(\frac{\sum_{x=1}^{n} MV_{(R_{jx},S_{ix})}}{n})}{N} \qquad 4.4$$

   where, $MV_{(R_{jx},S_{ix})}$ represents the match values of the QoS parameters of a certain requester-service pair, n is number of QoS parameters, and N is the number of requester-service match. A detailed description of the match value concept is given in Section 4.4.1. The objective of the PSI algorithm is to ***maximize*** $f(MV_{(R_{jx},S_{ix})}, N, n)$.

2. **Initialization:** The second step of the algorithm is to randomly initialize all the particles in the swarm, with the list of requesters and services. This step includes randomly matching the requesters and the services, and generating a match score for the same to initialize the population of particle. Further, the initial velocities of the particle is randomly determined between the range of *-max velocity* and *+max velocity*, and then the personal best P_{best} positions for all the particles are initialized. The swarm is then evaluated to give the initial global best guide G_{best}, based on the objective function of the problem. In this case, the G_{best} particle is the particle with the *maximum* match score. The objective function in this case is calculated using the equation mentioned in Equation 4.4. Also, variable named *bestpos* is defined to keep a track of the G_{best} particle in the swarm. A brief pseudo code of PSI selection process can be seen in Algorithm 4.

3. **Core steps of PSI Algorithm:** This step is the heart of the PSO algorithm. A counter is set to check the maximum number of iterations has been reached. If the program has reached

the maximum number of iterations, it stops. Otherwise, it executes the following steps:

- Evaluate the objective function $f(MS_{(R_j,S_i)})$ for all the particles in the swarm.
- Update the personal best (P_{best}) for all the particles.
- Update the global best (G_{best}) of the swarm. The global best selection is done based on the crowding distance concept, which is explained in the following paragraphs.
- Update the velocity of each particle, based on the following equation:

$$V[i] = W \times V[i] + R_1 \times (P_{best} - P[i]) + R_2 \times (G_{best} - P[i]) \qquad 4.5$$

  In Equation 4.5, $V[i]$ is the current velocity of the particle $P[i]$; W is the inertia weight is between the range of 0.5 and 0.3. and R_1 and R_2, are the acceleration constant fixed at 2. Finally, P_{best} and G_{best} are the personal and the global best guide of particle $P[i]$ respectively. The value W, R_1 and R_2, has an important role in the velocity equation. The inertia weight W controls the impact of the previous velocities on the current velocity of the particle. It also influences the trade-off between global and local optima, i.e., wide area and narrowed area exploration capabilities of the particles. The inertia weight W is varied between the range of 0.5 and 0.3 over the first 2000 iterations. A greater value of W during the initial iterations avoids getting stuck in the local optima. Whereas a smaller value for the later iterations, i.e., when the particles are closer to the global optima, ensures a better local area search. The role of R_1, and R_2 are similar to that of W, i.e., both are responsible to influence the impact of P_{best} and G_{best} respectively, on the velocity of the particle.
- Update the particle position, P[i]=P[i]+V[i].

This step is continued until the maximum number of iterations is reached.

4. **Particle Swarm Parameter Definition:** In this section we define the PSO parameters, such as the particle size, no. of iterations, type of problem - minimization or maximization, etc. In this research the particle parameters are fixed as follows:

   - Maximum number of iterations: 5000;
   - Number of particles: 25;
   - Acceleration constant 1 (local best influence): 2;
   - Acceleration constant 2 (global best influence): 2;
   - Initial inertial weight, default: 0.5;
   - Final inertial weight, default: 0.3;
   - Problem description: maximization;

- Maximum particle velocity: 1;

These values are taken based on previous experiments conducted using PSO, and some are also suggested by the authors of [91, 92]. The next step of the algorithm involves initializing the particle swarm.

5. **Crowding Distance calculation:** The best global guide in the swarm is selected based on the crowding distance concept, which is one of the most crucial steps in the PSI service selection algorithm. It mainly affects both the convergence capability of the algorithm, based on the personal best solutions of the swarm. In this algorithm, a matrix stores personal best solutions found in the previous iterations for each particle in a descending order. Logically, because it is a maximization problem, the solution with the maximum value in the matrix can be chosen as the global best guide of the particles in the swarm. However, in order to ensure that the particles in the population converge towards the global optima and not a local optima, in PSI algorithm, the global best guide of the particles is randomly selected from the top 20% of the personal best solution matrix. In this case, the number of particles in the swarm is 25, the global best guide is randomly selected in each iteration from the top 5 maximum personal best solution in the sorted matrix. Selecting different global guides for each generation from the specified top part of the sorted list, allows the particles in the population, to diversify their search for the global best particle, hence, gives a better solution.

Algorithm 3: PSI Selection Algorithm

Data: R[ ],S[ ],P[ ],OBJ-FUNC
Result: MS,Time
begin
  $t \longleftarrow 0$ (Initialize iteration counter)
  for $1 \leq i \leq n$ **do**
    Randomly initialize P[i]
    Initialize V[i] =0
    Evaluate P[i]
    Set P_{best} *of P[i]*
    Set G_{best} *of P[n]*
  while !*maxiterations* **do**
    for $1 \leq i \leq n$ **do**
      Evaluates the given Objective Function
      Set P_{best} *of P[i]*
      Calculate crowding distance (CD) for $P_{best}[i]$
      $A \longleftarrow SortAinDescendingorderofCD$
      Select G_{best} *from top 20% of A*
      Set G_{best} *of P[i]*
      Update Velocity for P[i]
      $V[i] = W \times V[i] + R_1 \times (P_{best} - P[i]) + R_2 \times (G_{best} - P[i])$
      Update Position for P[i]
      $Pos[i] = V[i] + Curr\_Pos[i]$
    $t \longleftarrow t + 1$
  return G_{best}
end

The steps mentioned can be seen above in the pseudo code of the PSI Algorithm in 3. The actual source code for PSI Algorithm can be found in Appendix A, Section A.2. The program iterates until the maximum number of iterations has been reached. Then the algorithm stops, and returns the global best match score derived from the objective function of this problem. In this research the MATLAB PSO Toolbox mentioned in [93] is modified and implemented to perform the experiments and the evaluations. The PSO Toolbox has been modified to fit the Grid service selection problem, and also to accommodate the MOPSO features, such as selecting the global best guide, the inclusion of crowding distance concept, etc. It is important to note that, unlike as mentioned in [1, 90], no mutation and/or crossover operators have been included in this algorithm, as in this problem the number of QoS parameters, and the total number of requesters and services in the pool is fixed, and cannot change over a period of time.

4.5 Munkres Assignment Problem Algorithm

A brief introduction of the Assignment Problem Algorithm is given in this section. The problem of selecting appropriate services for a set of requesters, from the pool of available services can also be defined as a problem of assigning the requesters to the services in an efficient way. Therefore, an assignment problem algorithm can also be used to solve this problem. Hence, the accuracy of the proposed algorithms are compared using a standard assignment problem algorithm. The assignment problem algorithm returns an optimal solution to the problem. A brief description of the assignment algorithm used to compare the proposed algorithms is given in the following paragraphs. The assignment problem usually consists of assigning n tasks to m agents, based on some cost criteria. The problem in this research is very similar to this, as there are n requesters to be assigned to m services, based on the match scores of each requester-service pairs. Hence, the assignment problem algorithm serves as a suitable benchmark to test the accuracy of the proposed algorithms. There are different variations of the assignment problem algorithm in the literature, mentioned in [94, 95, 96, 97, 98, 99]. The assignment problem algorithm used for this research is the one proposed by James Munkres in 1957, which is also known as the Munkres Algorithm [100, 101, 102, 21]. The assignment problem as formally defined by Munkres in 1957 [102] is: *"Let r_{ij} be a performance ratings for a man M_i for job J_i. A set of elements of a matrix are said to be independent if no two of them lie in the same line (the word "line" applies both to the rows and to the column of a matrix). One wishes to choose a set of n independent elements of the matrix (r_{ij}) so that the sum of the element is minimum."* Similarly, the problem of service selection on the Grid, in terms of an assignment problem can be defined as: An $n \times m$ requester-service matrix, representing the match scores of each requester with every other service. The match score matrix is the matrix represented in Equation 4.4 defined earlier in this chapter, where each element of

the matrix represents the match score for an individual requester-service pair. Munkres algorithm works on this matrix, to assign the requests to services so as to achieve an overall maximum total match score. It is important to note that the scope of this research considers that, each service can serve only one request and each request can be assigned to only one service. Hence, the assignments constitute an independent set of the requester-service match. Although, Munkres algorithm is said to guarantee optimality, it has a time complexity of O(n^3) to solve the problem, hence the algorithm is not very time efficient. A detailed discussion on this is given in the following chapter. A detailed description of the steps of the Munkres algorithm [102] is given below:

1. **Step 1:** A $n \times m$ matrix called the match score matrix in which each element represents the match score of assigning one of n requests to one of m services is considered;

2. **Step 2:** For each row of this matrix, the largest value of the match score is found, and is subtracted from every element in the row. The absolute values are taken. Proceed to Step 3;

3. **Step 3:** A zero (Z) is searched for in the resulting matrix. If there is no *starred zero* in the row or column, the Zero is starred $Z*$. This is repeated for each row in the matrix. Proceed to Step 4;

4. **Step 4:** Then each column containing a starred zero is covered. If K columns are covered, the starred zeros describe a complete set of unique assignments. If this is the case, the algorithm goes to Step 8, otherwise, go to Step 5.

5. **Step 5:** In this step, a non covered zero in the matrix is found, and is primed. If there is no starred zero in the row containing a primed zero, the algorithm jumps to Step 6. Otherwise, this row is covered and the column containing the starred zero is uncovered. This continues until there are no uncovered zeros left. Finally, the smallest uncovered value is saved and the algorithm switches to Step 7.

6. **Step 6:** A series of alternating primed and starred zeros are constructed in this step. Let Z_0 represent the uncovered primed zero found in Step 5. Let Z_1 denote the starred zero in the column of Z_0. Let Z_2 denote the primed zero in the row of Z_1. This series of alternating primed and starred zero construction is continued until the series terminates at a primed zero that has no starred zero in its column. Then, each of the starred zero of the series is un-starred, and each of the primed zero of the series is starred, and finally all the primes are erased and every line in the matrix is uncovered. And the algorithm loops back to Step 4.

7. **Step 7:** The value found in Step 5 is added to every element of each covered row, and is subtracted from every element of each uncovered column. The algorithm loops back to Step 5 without altering any stars, primes, or covered lines.

8. **Step 8:** The assignment pairs are indicated by the positions of the starred zeros in the match score matrix. If MS(i,j) is a starred zero, then the element associated with row i is assigned to the element associated with column j, is added and then divided by the total number of matched to get the overall match score.

The source code for this algorithm can be found in Appendix A, Section A.3.

4.6 Summary

This chapter discussed and analyzed the issues that were considered while designing the proposed selection algorithm. It also explained how these issues can make the Grid a more efficient service oriented architecture. The issues primarily emphasized the need of an appropriate service selection algorithm that should be able to match the requests and the services efficiently and accurately, under various scenarios. Furthermore, the two main concepts, the QoS metric and the Evolutionary Algorithm, on which the proposed algorithms are based on were thoroughly discussed, followed by a detailed description of the proposed algorithm: CSS selection algorithm and PSI selection algorithms was presented. It also analyzed and explained the reasons behind introducing the crowding distance concept while implementing the multi-objective particle swarm optimization technique. The chapter also introduced the service selector architecture, providing the reader with a brief overview of how the service selection algorithms are integrated within the Grid. Finally, a brief introduction to the Munkres assignment algorithm was given, as the algorithm is used to verify the accuracy of the proposed algorithms. The next chapter, presents the measurement setup, the experiment results, and a detailed analysis of both algorithms based on the results achieved by the algorithms.

Chapter 5

Experiments and Evaluations

The solution to the service selection problem is based on the fact that, the relevant service characteristics that are of interest for the user can be vividly represented, as specific QoS parameters. Also, this research addresses the issue of giving the requesters the flexibility to assign a weight value, W to the QoS parameters. The weight W provided by the requesters (mentioned in Chapter 4), for each QoS parameters reflects the degree of preference the requester has put for each of them. The value of W ranges between 0 and 1, and a value closer to 1, represents higher preference. Additionally, an evaluation method for measuring the degree to which the services fulfill the user expectations has also been incorporated. This is done by calculating the average match score for the requester-service set, and a higher match score indicates a better match, hence indicates a higher satisfaction rate from the requester's point of view. In this chapter, the experiments and the evaluation to study the efficiency of the proposed service selecting algorithms addressing the above criteria is presented. The results presented reflects the efficiency of both algorithms to assign requesters to services based on the QoS metric. The goal is to provide users with an efficient service selection mechanism, which allows them to express their requirements and preferences about the service providers when submitting a request to the Grid. This chapter discusses the methodology, implementation, experiments and the analysis done to evaluate the efficiency and the performance of the proposed algorithms, and has been categorized into following subsections:

- **Program Structure and Data Flow Analysis:** The section briefly explains the data flow within the algorithms. The program code for each of the algorithm can be found in Appendix A.
- **Assumptions:** This section discusses the assumptions, based on which the algorithms are designed, such as the type of services, the QoS parameters, number of requesters considered.
- **Measurement Setup:** A brief description of the experimental setup and the measurements done to analyze the efficiency of the proposed algorithm is explained in this section.
- **Results and Evaluation:** A detailed data analysis is presented in this section. The proposed algorithm are individually as well as combinedly compared to evaluate the performance of the both.

5.1 Program Structure and Data Flow Analysis

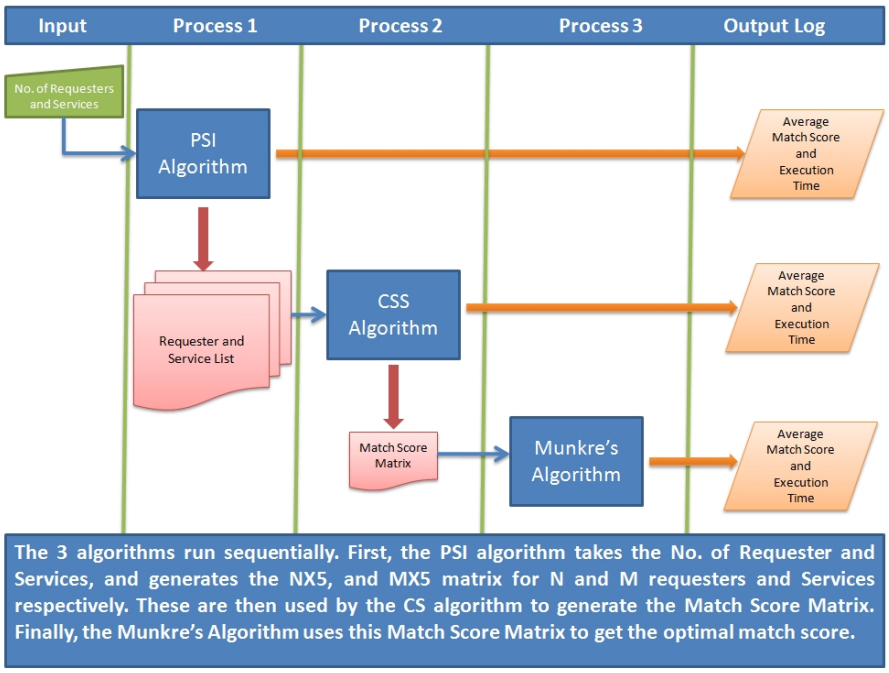

Figure 5.1: Process and Data Flow Chart.

The proposed algorithms are tested on a simulated data set that has been randomly generated to run the experiments. In order to generate these data sets, first the program requests for the number of requesters and services as either a run time argument or as a function argument. These number are then passed to the PSI function to generate the Requester-Service matrix. The PSI function generates a random $N \times 5$ requester matrix, and a $M \times 5$ service matrix where N is number of requesters and M is the number of services on the Grid. The algorithm writes the requester, and the service provider matrix in two separate files, which is later read by the CSS algorithm. The PSI function then processes the data using the steps mentioned in Algorithm 4, in Section 4.4.2 of the Chapter 4, and outputs the average match score and the execution time in the output log. The CSS algorithm function is called next. The CSS function takes the requester and service matrix as inputs, and runs the CSS algorithm on it. After the algorithm has finished processing the data it generates the match score matrix MS(as mentioned in Chapter 4, Section 4.4.1), and writes it to an output file. It also outputs the average match score and the execution time in the output log. Finally, the Munkres function reads the match score matrix as an input and processes it to generate the optimal match score. This method ensures that all three algorithms are working on the same data set, and generates the match scores based on a consistent input.

5.2 Assumptions

The design and implementation of the proposed algorithms in this research are based on the following assumptions:

- **Type of requests:** A user request can involve the following types of requirements, such as - a user can require the execution of a single request; a user may specify the coordinated execution of a set of services, typically modeled as a workflow; the user can also specify requirements for one or more services that should serve the request; or the user may also specify requirements on the network connectivity of the resource executing the service. This research solely focuses on the users requesting the execution of a *single* service. Although the request is for a single type of service, the users may specify different parameters of the QoS metric, which needs to be presented at the request submission time.

- **Type of services:** In this research only a single type of service has been considered. The proposed algorithms works on multiple copies of a single type of service. Although, the request is a single type of service, each service provider has a unique set of QoS parameter.

- **Number of available requesters/services:** The number of available services at a certain time t_x is always fixed. It is assumed that the information about the available services for a certain period of time is spooled into a registry which is made available to the requesters during the selection process. The information is updated periodically, and not dynamically during a ongoing selection process. Similarly, the number of incoming requesters at a certain time t_y is always fixed. It is assumed at the incoming requesters from time t_1 to t_2 are stored in a registry and then made available to the requester-service matcher mentioned in Figure 4.6. The pool is updated periodically, and not dynamically during the service selection process.

- **Service selection process:** The proposed algorithm only selects the appropriate service from the pool of available services for a particular requester based on the QoS parameters requested by them. No scheduling is done in this process. The selection problem discussed in this research is completely independent of the task of a scheduler.

- **QoS Parameters:** In this research only five QoS parameters has been used for the selection process namely, time, cost, availability, reliability and reputation. Although, this list is not exhaustive, it reflects the most commonly considered QoS parameter set mentioned in the literature. As mentioned earlier, a variable W is also multiplied with these QoS parameters, which represents the weight value of the corresponding QoS parameter.

These assumptions have been considered while designing the proposed service selection algorithms. These also hold true for all the experimental setups and the evaluations done to test the efficiency and the feasibility of the algorithms. Although, PSO algorithms are flexible and can be modified to fit any optimization problem, the consideration of scenarios apart from the ones discussed above is beyond the scope of this research.

5.3 Measurement Setup

PSI, CSS and the Munkres algorithm, are implemented using MATLAB 7.5.0 using the MATLAB editor version R2007b. Various sets of measurements are performed, to test the scalability, accuracy, and feasibility of the algorithms. Each measurement point represents the average of 10 runs of the respective experiments. The measurement criteria are designed mainly to evaluate the execution time, the average match score under various requester-service matching scenarios. The PSI and the CSS algorithm, are also tested for scalability, and for its performance under the scenarios where the requesters are competing for similar kinds of requests. The experiments were run on an DELL XPS M1330 laptop, with Intel Centrino Processor, 4 GB RAM, and 250 GB HDD. A brief description of all measurements performed is given in the following section.

5.3.1 Methodology

The methodology for running experiments for the algorithms included, first evaluating the algorithms individually and then comparing them based on the prior evaluation results. Each algorithm is evaluated based on three different requester-service pair settings, namely - 500-500, 1000-500, 500-1000. These settings help to analyze the performance of the algorithms under conditions where there are an equal number of requesters and services, and also when the number of requesters are more or less than the available services. The third criteria, where the number of available services is greater than the number of requesters, the optimization required is less than the other two criteria, unless most of the requesters in this are competing for similar services. The evaluation criteria of the algorithms are based on three major sets of measurements: 1) the average match score achieved for the given requester-service set, 2) the execution time of both algorithms, and 3) the scalability. Additionally, for both algorithms, the QoS parameters of services and the requester ranges between 0 and 1. Requesters are also allowed to assign a random weight, ranging between 0 and 1, to the QoS metric in order to provide a preference value to each of the QoS parameters. The QoS parameters for all the services and requesters are generated randomly. Furthermore, for evaluating the performance of both algorithms the fraction of requesters competing for similar services is varied, by generating more requester with QoS parameter within a close range. A detailed description of each of the algorithm evaluation criteria is given below.

Measurements common to both algorithms:

In this section a brief description of all the measurements performed to evaluate both the CSS and the PSI algorithm, and a reasoning of the same is given. For the measurements of the PSI algorithm, the number of particles were fixed to 25, and the maximum number of iterations were fixed to 5000. The two random variables R1 and R2, in the velocity equation (in Equation 3.4) of the algorithm was fixed to 2 and 2 respectively, and the inertia weight W was varied from 0.5 to 0.3 over 1500 iterations. Some additional measurements performed on the PSI algorithm are given in the following section.

1. **Average Match Score:** The average match score is measured for both algorithms for three different scenarios of 500-500, 500-1000 and, 1000-500 requester-service pair settings.

2. **Execution Time:** The execution time of both algorithms is measured by varying the number of requester-service pair from 100-100 requester-service pair to 1000-1000 requester-service pair.

3. **Scalability:** The scalability of both algorithms is measured, by gradually increasing the number of incoming requesters, while keeping the number of services fixed.

4. **Effect of similar requests on the execution time:** The fraction of requesters submitting similar requests are varied from 5% to 95% to see its effect on the execution times of the algorithm.

5. **Effect of similar requests on average match score:** The fraction of similar requesters submitting the requests are varied from 5% to 95% to see its effect on the average match score of the algorithms.

Measurements specific to PSI Algorithm

Some additional measurements specific to the PSI are given as follows:

1. **Effect of varying particle size on execution time:** The particle size of the PSI is varied, to investigate its effect on the execution time of the algorithm.

2. **Effect of varying particle size on average match score:** The effect of varying particle size is also investigated on the average match score obtained by the algorithm.

Comparison of the proposed algorithms with Munkres Algorithm:

In this section the performance and the accuracy of both algorithms is compared with the Munkres Assignment Algorithm. The comparison are based on the following criteria:

1. **Comparison of the Average Match Score:** The average match score achieved for both algorithms are compared with the average match score obtained by the Munkres assignment algorithm, under the three different requester-service settings mentioned earlier. This comparison gives a fair idea about the accuracy of the proposed algorithms.

2. **Comparison of the Average Execution Time:** In this section, the average execution times of the proposed algorithms are compared with the average execution time of the Munkres assignment algorithm, to check for the efficiency of both algorithms under the three different requester-service settings.

The following section discusses and analyzes the results achieved after investigating the performance of both algorithms based on the above mentioned evaluation criteria.

5.4 Results and Evaluations

In this section an elaborate description of the performance evaluations of the algorithms are presented. First, the match score achieved by each of them is evaluated, and then the execution time of both is presented and analyzed. Finally, the scalability of both algorithms are evaluated.

5.4.1 Average Match Score Evaluation

Figure 5.2, 5.3, and 5.4 plots the average match score against the number of iterations, keeping the requester-service pairs fixed to 500-500, 500-1000 and 1000-500 respectively for the PSI algorithm. From these three figures it can be concluded that the algorithm performs efficiently by achieving an average match scores of 0.87, 0.94, and 0.70 approximately after 5000 iterations for 500-500, 500-1000, and 1000-500 requester-service pairs respectively. The global best values indicates the best average match score achieved by the global best guide in the swarm during that iteration. In this problem a higher global best value, indicates a better and closer, requester-service match score. From the figures, it can also be concluded that the increase in the global best values in the initial iterations, as the swarm particles tries to converge to the global optima, is more rapid than in the later ones. Also, the value of the global best for all three cases has a stabilizing trend after approximately 3000 iterations. The figures also represents the converging behaviors for the three requester-provider settings. It can be seen that for 500-500 requester-service pair settings the average match score varies between 0.850 and almost 0.873. While, for 500-1000 and 1000-500 requester-service pair settings it converges from 0.70 to 0.94, and 0.48 to 0.70 approximately. Initially, for all the requester-service pair scenarios the average match score has a sharp increase and then it gradually stabilizes as the swarm converges towards the global best solution. Also, in the 500-1000 requester-provider settings, the average match score is significantly better than 1000-500 and 500-500 requester-service scenario, the reason for this is explained in the next few paragraphs.

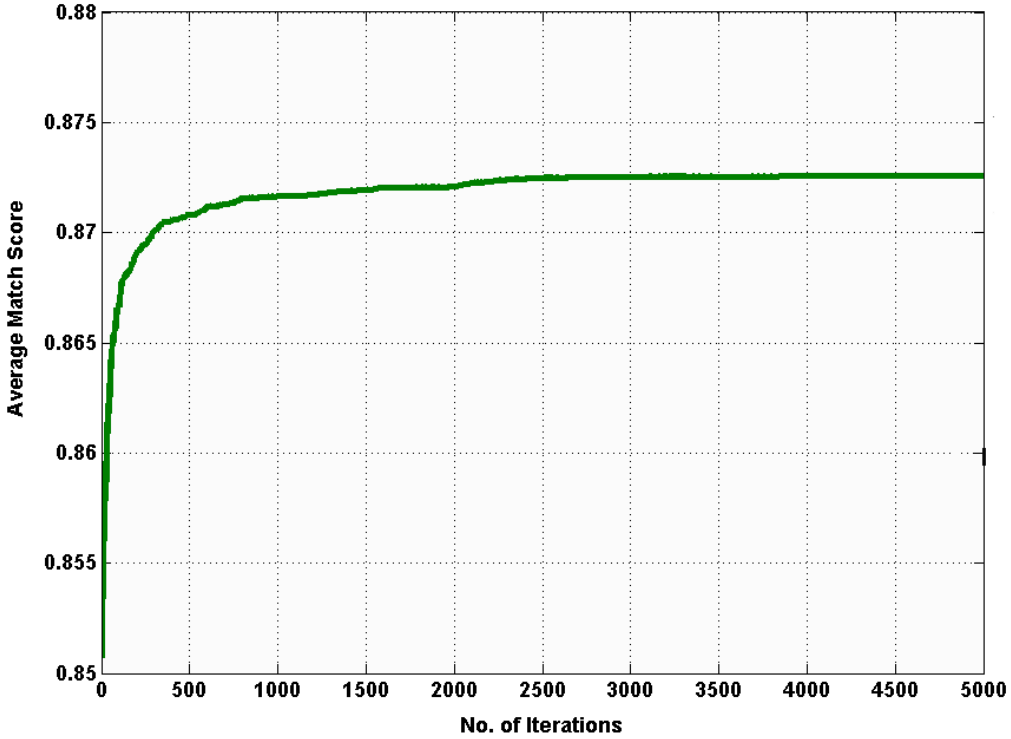

Figure 5.2: Average Match Score vs. No. of Iterations of PSI algorithm for 500-500 requester-service pair.

Figure 5.5, presents the average values of the match score for 500-500, 1000-500, and 500-1000 requester-service pairs for the CSS algorithm. From this graph it can be concluded that the algorithm achieves a match scores of 0.66, 0.58 and 0.86 for 500-500, 1000-500, and 500-1000 requester-service pairs respectively. Comparing the average match scores obtained by both algorithms we see that the PSI algorithm clearly achieves a higher, hence better average requester-service match score than the CSS algorithm. A graphical representation of this comparison is shown in Figure 5.6. From the graph it can be inferred that the PSI Algorithm outperforms the CSS algorithm with a significant margin in the case of 500-500 requester-service settings. Also, for 500-1000 and 1000-500 requester-service pair settings, the average match scores achieved by both are marginally different, and PSI outperforms CSS with a considerably higher match score. Hence, PSI achieves a better overall match score for all the scenarios. Additionally, Figure 5.6 represents the comparison of the CSS and PSI algorithm. It is important to note that for the measurements in Figure 5.6 the PSI algorithm had gone through 5000 iterations. A more detailed comparison of the average match scores achieved by both algorithms is presented in a tabular format in Table 5.1. It shows that for PSI the average match scores for 500-500, 500-1000, and 1000-500 requester-service pairs stabilizes around 0.87, 0.94 and 0.70 respectively at 5000 iterations, while the average match scores for the CSS algorithm for the three scenarios is 0.66, 0.86 and 0.58 respectively.

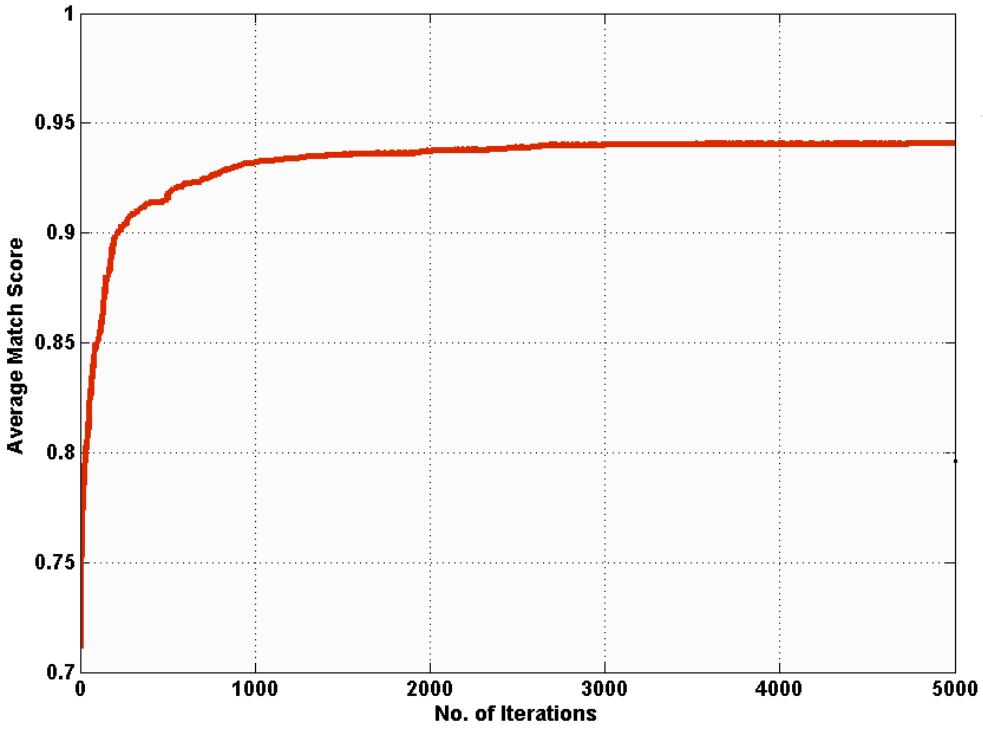

Figure 5.3: Average Match Score vs. No. of Iterations of PSI algorithm for 500-1000 requester-service pair.

The following conclusions can be drawn from the results obtained.

- The average match score obtained by the PSI Algorithm gives an optimized match score for all the three scenarios and it ranges between 0.70 and 0.94, unlike in the case of the CSS algorithm, where the match score ranges between 0.58 to 0.86 which is considerably lower than PSI. The CSS algorithm fails to optimize the matching process efficiently as opposed to PSI algorithm, when there are equal number of requesters and services competing to get itself assigned to an appropriate service.

- Furthermore, looking at all the three scenarios it can be concluded that at a given time the number of successful matches equals to either the number of available services or the number of incoming requests, whichever has a lower value. In this problem, for all the three cases the number of successful matches are essentially 500. An efficient selection algorithm would thereby be able to find a good distribution of optimized solutions for all the three cases, instead of given better result for just one case, as compared to others. This goal is achieved by the PSI algorithm. The optimization process is the toughest in case of equal number of requester-service pair, as the requesters are competing amongst themselves to get a better matched service. In this scenario the PSI algorithm significantly outperforms the

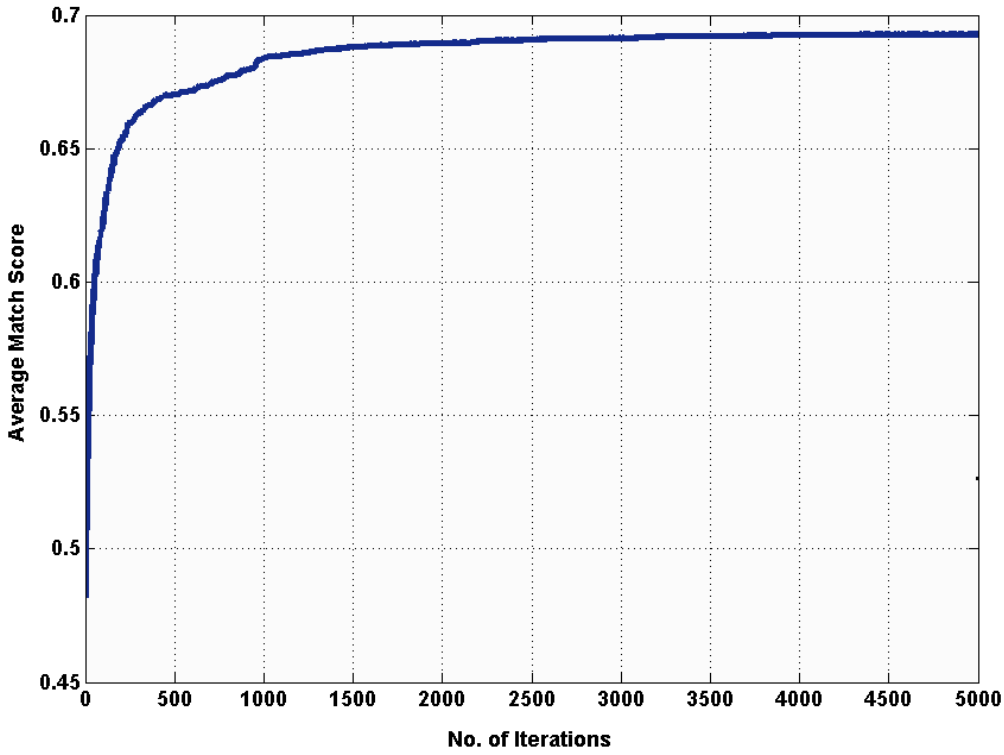

Figure 5.4: Average Match Score vs. No. of Iterations of PSI algorithm for 1000-500 requester-service pair.

CSS algorithm by achieving a 24.14% higher match score, and hence giving a closer match between the requester-service pair.

- In the case of 500-1000 requester-service pair scenario both the algorithms achieve the highest match score when compared to the other two scenarios. This is because, in this case the number of available services are more than the number of requesters. This allows the requesters to have more than one service provider available to each of them to choose from, and hence seeks less optimization in the selection process. In this case CSS algorithm achieves an average match score of 0.86 and PSI on the other hand converges at 0.95 average match score after 5000 iterations. The PSI algorithm outperforms the CSS by 9.47%, and achieves a better solution during the selection process.

- Also, in case of the 1000-500 requester-service pair scenario both the algorithms achieve the lowest match score when compared to the other two scenarios. This is because, in this case the number of requesters seeking the services on the Grid are twice as much as the number of available services. As mentioned earlier this research has considered static allocation of services, i.e., the number of available services in the pool is always fixed. Hence, in this case the number requesters achieving a successful match is only 500, even though there are 1000

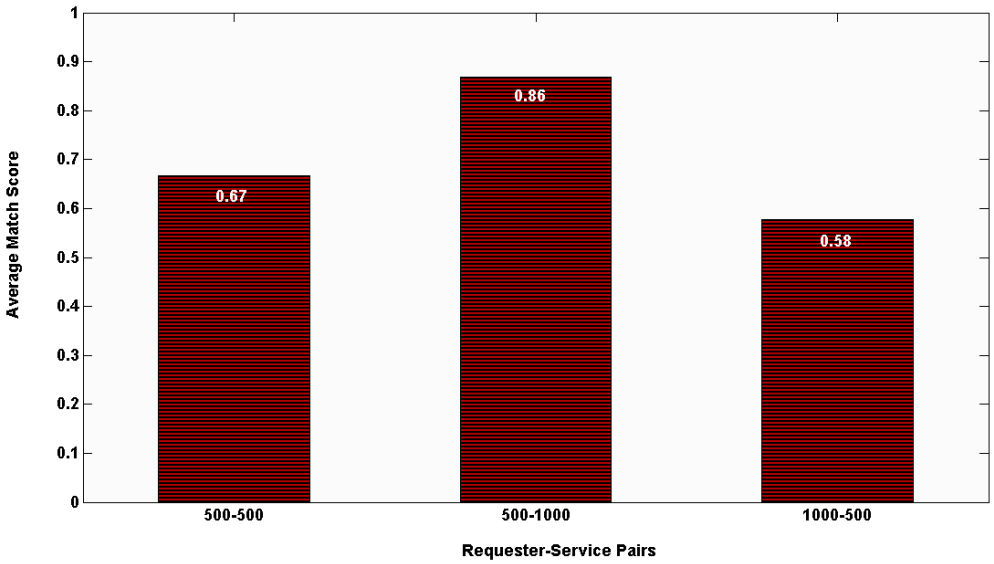

Figure 5.5: Average match score for requester service pairs of CSS algorithm.

requester in the requester pool.

- The average match score percentage increase of the PSI and CSS algorithm for 500-1000, 1000-500 and 500-500 requester-service pair scenario are 9.47%, 17.14% and 24.14%. Considering all the above cases, the average match score achieved by the PSI algorithm and CSS algorithm is, 0.84 and 0.70 respectively, whereby PSI achieving 16.67% higher match score than CSS.

Table 5.1: Comparison of match score

Requester-Service pairs	**Average Match Score**	
	CSS	**PSI**
500-500	0.66	0.87
500-1000	0.86	0.94
1000-500	0.58	0.70

Also, considering that the best optimized match score can be one, and a higher match score value indicates a more efficient and closer match, the experimental results confirm that the average match score achieved for all three scenarios, in case of PSI at 5000 iterations is closer to the optimized solution than the CSS algorithm. These results also strongly support the hypothesis that using the PSI algorithm to optimize the process of assigning requesters and services on the Grid, achieves a

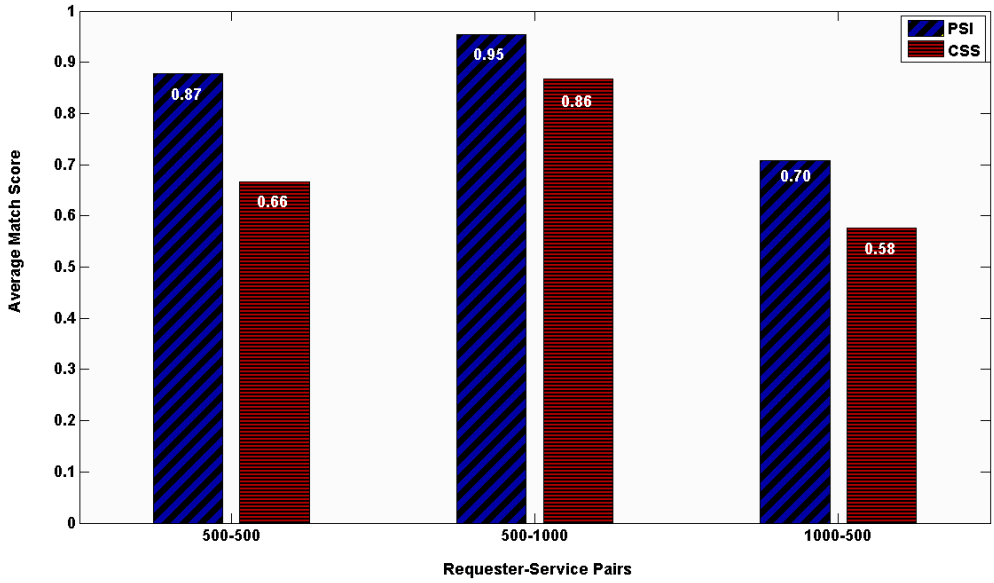

Figure 5.6: Average match score for requester-service pairs of CSS and PSI algorithm (at 5000 iterations).

high accuracy in terms of better and unbiased matches, in comparison to the CSS algorithm. Table 5.1 summarizes the different match scores achieved by both algorithms.

5.4.2 Execution Time Evaluation

The next set of experiments compare the execution time for the two algorithms. Figure 5.7 shows the execution time in seconds versus the number of different requester-service pairs for the PSI algorithm, keeping the maximum number of iterations fixed to 5000. To obtain an equally distributed evaluation, the number of requester-services pairs is varied from 100 requester-service pairs to 1000 requester-service pairs.

From the above graph it can be seen that, in case of the PSI algorithm, 100, 500 and 1000 requester-service pairs resulted in execution times of 5.96sec, 34.15sec and 132.38sec respectively. The average execution time of the PSI algorithm is calculated as 47.87sec, when the particle size and the maximum number of iterations is fixed to 25, and 5000 respectively. It can be seen that the execution time increases approximately by 20 times, when the number of requesters and services to be matched each is increased by 10 times. Figure 5.8 on the other hand, represents the execution time for the CSS algorithm. The execution times for 100, 500, and 1000 requester-service pairs resulted in 0.45sec, 12.23sec and 46.75sec respectively. The average execution time for the CSS algorithm is 18.48sec. The graph in Figure 5.8 indicates an increase in the execution time with the increase in the number of requester and services. However, the average execution time for the PSI algorithm is approximately 62.19% more than the CSS algorithm.

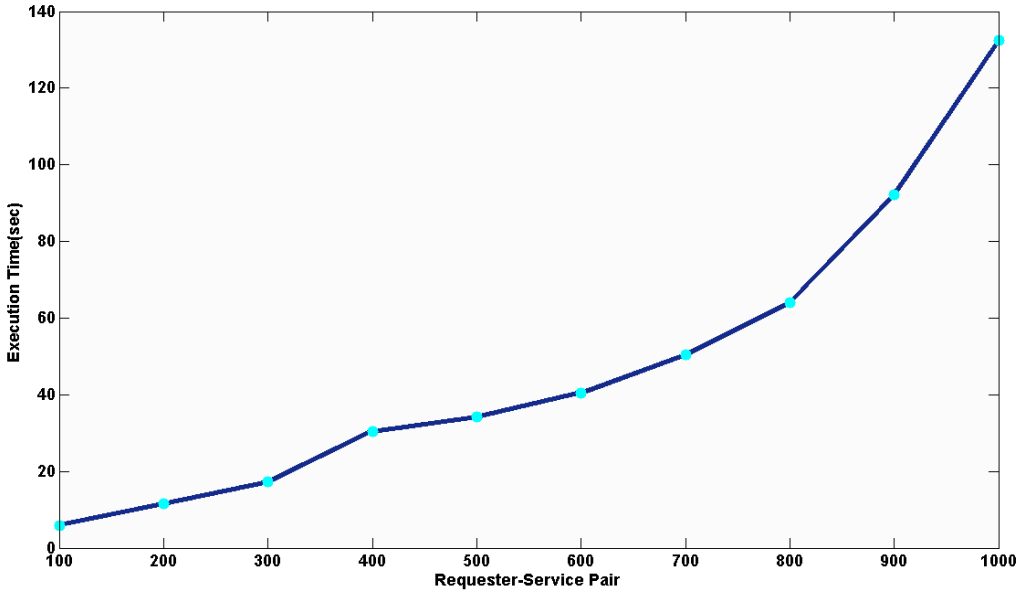

Figure 5.7: Execution time(sec) vs. Number of requesters-service pairs of PSI algorithm.

The execution time for the CSS algorithm is clearly better than for the PSI algorithm. A detailed analysis of the same is given in Table 5.2. The table represents the comparison of the execution time in seconds for 200, 400, 600, 800, 1000 requester-service pairs for both algorithms. From Table 5.2 it can be inferred that CSS is faster than the PSI algorithm. Also, it can be seen that the average execution time for the CSS algorithm is approximately 2.5 times faster than the PSI algorithm.

Table 5.2: Difference of execution time

Requester- Service pairs	**Execution time(sec)**	
	PSI	**CSS**
200-200	11.53	1.78
400-400	30.51	7.58
600-600	40.48	17.88
800-800	63.93	30.61
1000-1000	132.38	46.75

The difference in speed of the PSI and the CSS algorithm can be explained on the basis, that the PSI algorithm follows the rules of the swarm intelligence algorithm, versus the fact that CSS is based

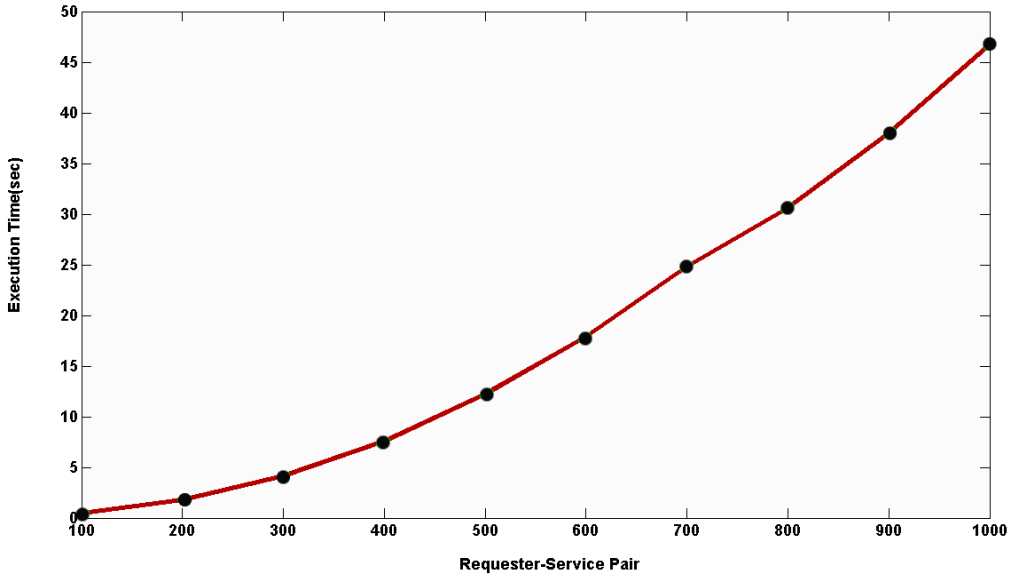

Figure 5.8: Execution time(sec) vs. Number of requester-service pairs of CSS algorithm.

on relatively simple mathematical calculations. In case of the PSI algorithm the potential solutions (in this case it is the requester-provider match pairs) are represented by the swarm particles, and the total execution time of this algorithm can be attributed to the fact that, of each iteration, the following computationally intensive calculations take place:

- In each iteration 25 swarm particles evaluate the objective function to find the global best particle;
- The selection of the global best is a non-trivial process, which entails sorting the solution in descending order and then selecting the global best from the top 10-20% of the swarm;
- Finally, the particle needs to keep track of the following variables for 5000 iterations:
  - Current Personal Best position and value;
  - Current Global Best position and value;
  - Previous Personal Best position and value;
  - Previous Global Best position and value;

All these steps are repeated to find the global best match scores for each particle, for every requester-service pair at until the maximum number of iterations has been reached. Whereas, in case of the CSS algorithm the requester-service pair matches, are based on a series of simple algebraic and arithmetic calculations e.g., calculating the match scores for each pair, and then assigning the services to the requester based on the constraint satisfaction problem solving approach. These steps in the CSS algorithm are not as computationally intensive as in the case of the PSI algorithm, and hence the former is faster.

5.4.3 Effect of Competitive Matching on the Average Match Score of the Algorithms

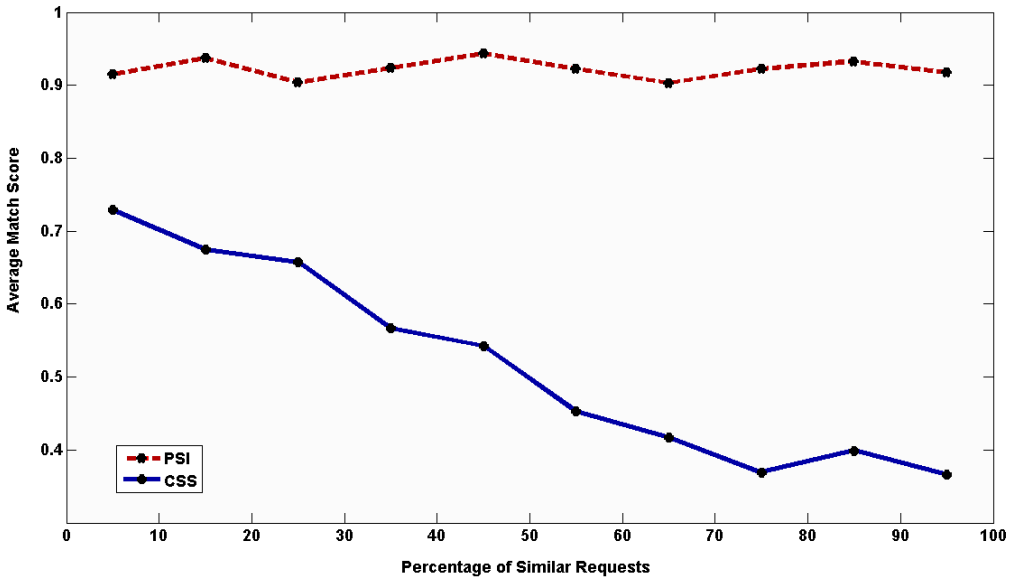

Figure 5.9: Effect of percentage change in similar request on the average match score.

This section discusses the effect of similar requests submitted by the requesters, on the average match score achieved by the proposed algorithms. The number of requester-services pairs for this evaluation is fixed to 500-500 for both algorithms. Furthermore, the particle size and the number of iterations for the PSI algorithm was fixed to 25 and 5000, respectively, and the values of R_1 and R_2 was fixed to 2 and 2, and W was varied from 0.5 to 0.3 over 2000 iterations. This evaluation criteria focuses on the accuracy of the match score achieved and the execution time taken, by the proposed algorithm, when a fraction of requesters has requested for similar kind of services. In the experiments the fraction of requesters requesting for similar services is varied from 5% to 95%. The QoS parameter for a fraction of requesters were generated randomly within a very close range, e.g., for 50% of the requester requesting similar services, 250 out of 500 requesters have a QoS parameters set within a very close range of 0.5 to 0.7. These similar requests would then compete for the available services from the pool. From Figure 5.9 it can be seen that the PSI algorithm has a better average match score as compared to the CSS algorithm, when the percentage of similar requests increases. Table 5.3 illustrates the average match score achieved by both algorithms, for cases when the percentage of similar requesters in the pool was fixed to 15%, 35%, 55%, 75%, and 95%.

From both, Table 5.3 and Figure 5.9, it can be inferred that there is a decreasing trend in the average match score, with the increase in the fraction of similar requesters for the CSS algorithm, while in case of the PSI algorithm the average match scores show a stable trend. The average match score obtained by PSI and CSS algorithm over the period, when the similarity percentage amongst

Table 5.3: Effect of similar requests on the average match score.

% of Similar Requests	Average Match Score	
	CSS	**PSI**
15%	0.67	0.94
35%	0.57	0.92
55%	0.45	0.92
75%	0.37	0.90
95%	0.37	0.92

the requesters was varied from 5% to 95% are, 0.92 and 0.52, respectively. The average match score obtained by the PSI algorithm is approximately 43.5% higher, and better than the average match score obtained by the CSS algorithm. The PSI algorithm clearly outperforms the CSS algorithm, by matching the requests more accurately with the services available in the pool. It does so because, in case of PSI, the algorithm optimizes the match procedure by adjusting the matches efficiently - i.e., in the event where there are large number of requesters demanding similar kinds of services, the algorithm seeks to optimize the *overall* match score of the requesters and the services, rather than just assigning a fraction of requesters to a group of closely matched services. On the other hand CSS algorithm unlike PSI, shows a decreasing trend in the average match score achieved. This is because, the CSS algorithm follows the "Greedy" approach and assigns the best services to some requesters while assigning worst to the remaining. Thereby, CSS algorithm compromises the overall match process. The performance of the CSS algorithm decreases with the increasing need of overall optimization in the service selection approach. Also, the stable trend in the average match score values, as the percentage of the similar requester increases, clearly reinstates the hypothesis, that PSI achieves a better match score in cases where more optimization is required.

5.4.4 Effect of Competitive Matching on the Execution Time of the Algorithm

In this section the effect of similar requests on the execution time is investigated. Requesters with similar requests were generated, by randomly generating their QoS parameter values within a close range, for e.g., 0.5. and 0.7. These closely generated requests compete for similar services. Under this scenario, the need for accuracy in the matching process becomes paramount. The other values of the parameters for this experiment are as follows: the number of requester-service pair is fixed

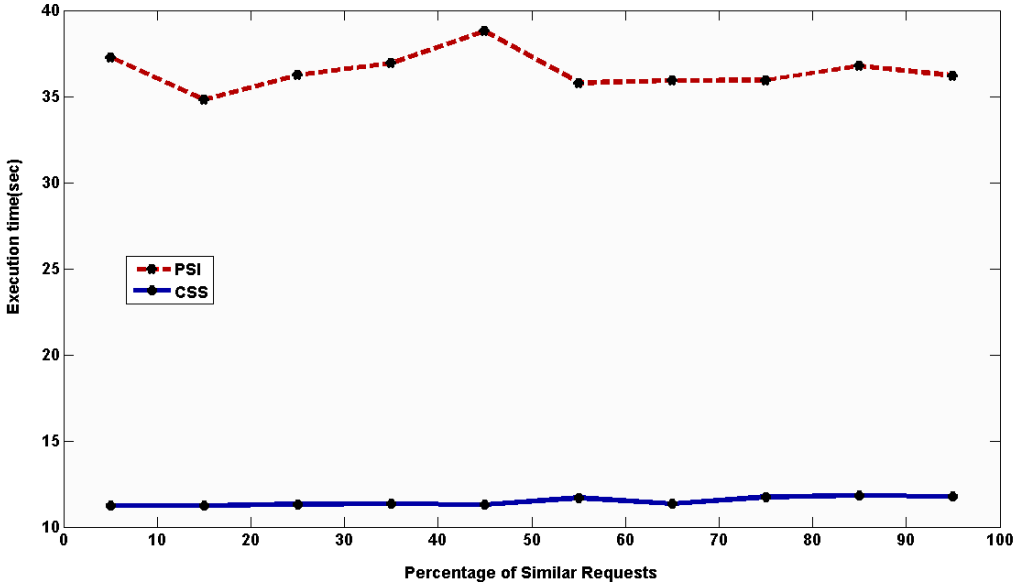

Figure 5.10: Effect of similar requests on the execution time.

to 500-500 for both algorithms, the number of particles for PSI is fixed to 25, and the number of iterations is 5000. The values for R_1 and R_2 is fixed to 2 and 2 respectively, and the value of weight inertia W is varied from 0.5 to 0.3 over 2000 iterations. Table 5.4 shows the execution time of both algorithms for 15%, 35%, 55%, 75% and 95% of similar requests.

Table 5.4: Effect of similar requests on the execution time (sec)

% of Similar Requests	Execution Time (sec)	
	CSS	**PSI**
15%	11.25	34.79
35%	11.34	36.94
55%	11.69	35.79
75%	11.73	35.95
95%	11.78	36.20

From Figure 5.10 shows the execution times of the proposed algorithms in seconds against the percentage of similar requesters. From the graph it can be inferred that for both the algorithms there is a minimal change in the execution time with the change in the fraction of similar requests. The average execution time is 11.48sec and 36.45sec for CSS and PSI respectively. From the above results, it can be inferred that, the execution times of the PSI algorithm is approximately 68.5%, more than the CSS algorithm. Hence, in this case the CSS algorithm is almost three times faster

than the PSI algorithm. The Figure 5.10, also represents a minimal change in the execution times of both the algorithms. This is due to the fact that, (as seen earlier in Section 5.4.2) the execution time of these algorithms are directly proportional to the number of requester-service pairs. And, in this case the number of requester service pairs remains fixed at 500, whereas only the fraction of the similar requests changes over the time. This fraction has no effect on the execution time. Hence, the execution time for both algorithms remains almost constant throughout. However, the overall increase in time for PSI algorithm when compared to CSS, can be attributed to the fact that with the increase in similar kinds of requests, the number of requesters competing for the same services would increase, hence more optimization is required to be done to match the requests to the services efficiently. This in turn increases the execution time of the algorithm, as the particles takes more time to evolve and converge to a better overall solution. However, it can be seen from Figure 5.9 that even though the execution time for the PSI is higher than the CSS algorithm, the accuracy of the average match score achieved in this is also significantly better, than the CSS algorithm. Summarizing, the match score achieved by the PSI algorithm is approximately 43.5% higher than the CSS algorithm. This proves that match score achieved by the PSI algorithm - in the case where higher percentage of the requesters are competing for similar services - is outstandingly better than the CSS algorithm. Also, the execution time of the PSI algorithm is 68.5% higher than the CSS, due to the evolutionary characteristics of the algorithm. In conclusion, it can be said that the PSI algorithm achieves a significantly better match score than CSS algorithm, for a marginally high execution time.

5.4.5 Scalability

In this section the scalability of both algorithms is evaluated by keeping the number of services available fixed to 500, and by increasing the number of incoming requesters linearly from 500 to 10000. The results of the same for both algorithms is discussed below.

In Figure 5.11 the effect of varying incoming requests on the average match score of both the algorithms can be seen. The requester and the service provider sets are generated randomly, and the average match score achieved by each algorithm is measured by changing the number of incoming requests from 500 to 10000, while keeping the number of services fixed to 500. This factor evaluates the accuracy of the algorithms, under the circumstances where there are a large number of incoming requests competing for a smaller number of services. From the Figure 5.11 it can be seen that the average match score for PSI algorithm varies from 0.93 to 0.58, when the number of incoming requests is changed from 500 to 10000 respectively. The average match score in this case is 0.77. Similarly, for the CSS algorithm from the figure it can be inferred that the average match score for this algorithm varies in between 0.61 to 0.25, yielding an average match score of 0.43. The match score achieved by PSI is 44.15% higher than the CSS algorithm. Hence, PSI achieves a better and

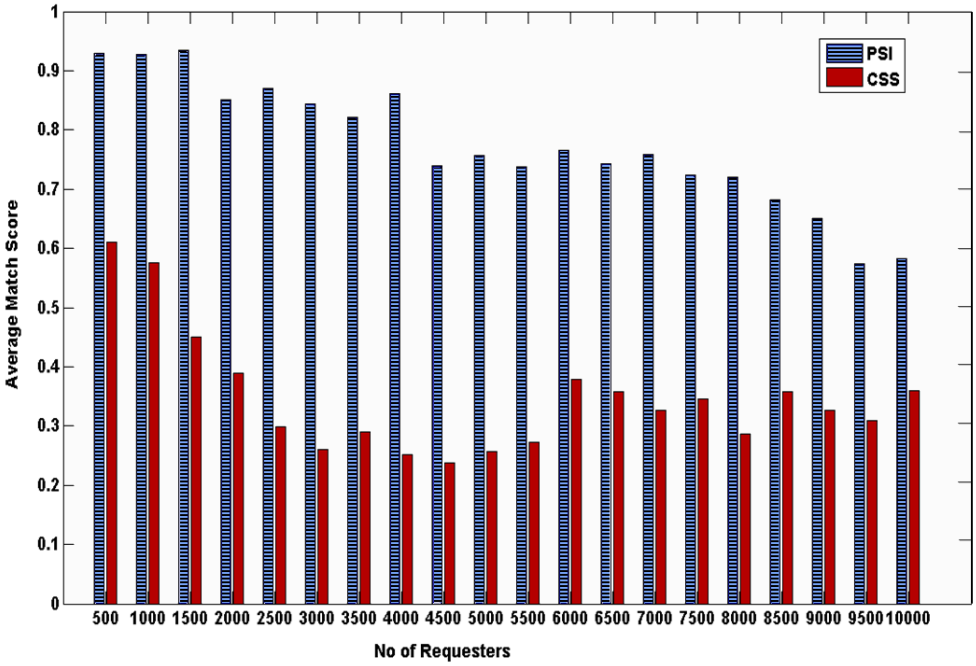

Figure 5.11: Scalability of the PSI algorithm.

closer match score in this case. It can also be seen that in case of CSS the match score oscillates between the range of 0.61 and 0.25. Whereas, the match score achieved by PSI algorithm decreases gradually from 0.92 to 0.58. Summarizing, the results indicate that, PSI achieves an overall higher match score, therefore responds better to scenarios which entails scalability. On the other hand, the average match score achieved by the CSS is not only varying, but is also significantly lower than PSI. The results obtained proves that the PSI algorithm, due to its high optimization capability, matches the requester-services more efficiently in the case of a higher number of incoming requests, than the CSS algorithm.

5.4.6 Effect of Particle Size on the Performance of PSI Algorithm

This section investigates the effect of particle size, on the average match score and the execution time of the PSI algorithm. The size of the particle plays a key role in any evolutionary algorithm, as it not only contributes to the optimization procedure but also plays an active role in the execution time of the algorithm. The experimental setup for this set of evaluation is as follows: the number of requesters-service pairs for both tests were fixed to 500-500; the number of iterations was fixed to 5000. The value of R_1 and R_2 was fixed to 2 and 2 respectively, and the value of the inertia weight W was varied from 0.5 to 0.3 over 2000 iterations. The following sections analyzes the effect of the particle size on the execution time and the average match score achieved by the algorithms,

respectively.

Execution Time

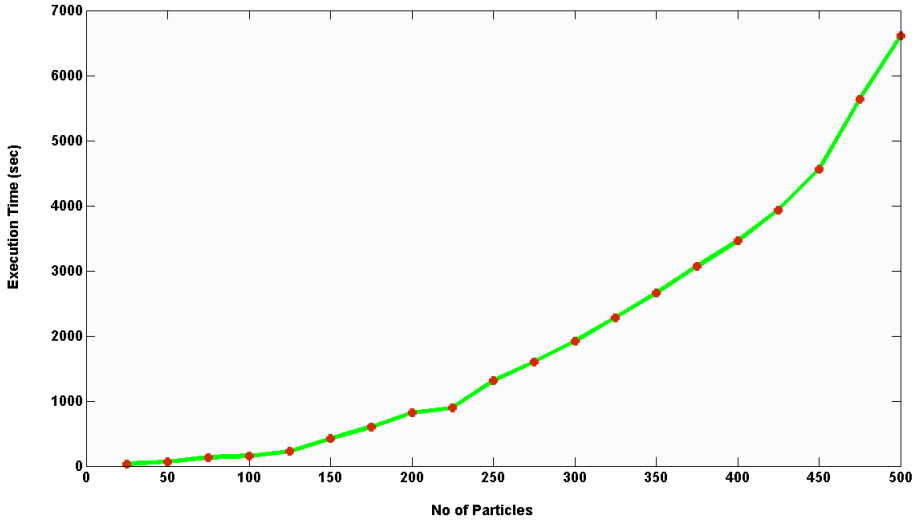

Figure 5.12: Effect of particle size on the execution time.

This section analyzes the effect of particle size on the execution time of the PSI algorithm. As seen from Figure 5.12, the execution time changes with the increase in the number of particles, for example, from the graph it can be seen that for particle size 100, 200, 300, 400, and 500 the execution time is approximately 156.80s, 815.43s, 1919.12s, 3457.61s, and 6611.09s respectively. The reason behind the increase in the execution time with the increase in the number of particles can be mainly attributed to the followings steps in the algorithm:

- With the increase in the number of particles the core steps of the algorithm increases proportionately. The core steps of the algorithm (mentioned in Chapter 4, Section 4.4.2), are now executed n times where n being the number of particles. Furthermore, these steps are iterated for the maximum number of iterations, which again adds to the execution time of the algorithm.

- Moreover, in this case, a greater number of particles needs to keep a track of its neighbors. And, that becomes a fairly large number as compared to the case when the number of particles in the swarm is less. Hence, it once again adds to the overall execution time of the algorithm.

From the above discussion, it can be concluded that due to the increase in the execution time with the increase in the number of particles it is suggested to keep the particle size to a relatively small value, which would balance the execution time and the average match score achieved. A good sample as discussed in the literature is between 25-50 particles depending on the problem.

Average match score

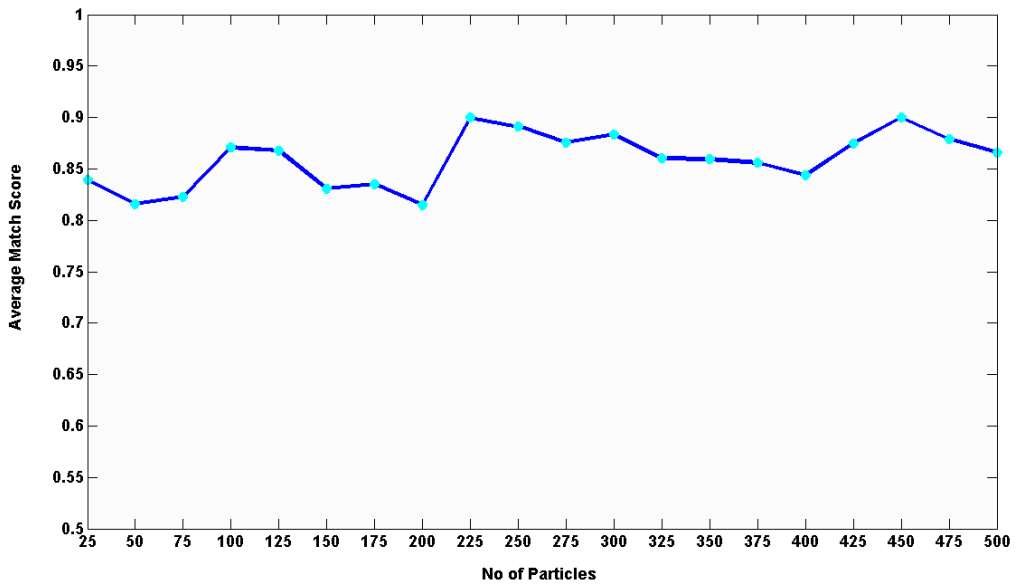

Figure 5.13: Effect of particle size on the average match score.

The effect of the particle size on the average match score of the PSI algorithm is analyzed in this section. As seen from Figure 5.13, the average match score does not vary a lot with the increase in the number of particles. The average match score achieved while varying the particle size from 25 to 500 is 0.86. The match score values varies between the range of -0.03 to +0.03 for the entire range, with the match score achieved for particle size of 100, 200, 300, 400 and 500 being 0.87, 0.81, 0.88, 0.84 and 0.86 respectively. It can be seen that the average match score remains almost unchanged, with the change in particle size. The average match score returned by PSI algorithm, when the particle size is changed from 25 to 500 is same as the average match score returned by the algorithm for 25 particles. Although, a slight increase in the match score is seen for a higher number of particles, in order to achieve that the execution time of the algorithm would have to be compromised significantly. Hence, it can be concluded that the size of particle has a considerably high effect on the execution time of the algorithm, while it does not make any significant difference to the average match score of the PSI algorithm.

5.4.7 Comparison of the Proposed Algorithm with Munkres Assignment Algorithm

In this section the two algorithms: PSI and CSS are compared with the Munkres assignment algorithm to check for the algorithms' accuracy, and efficiency. The first section discusses the

comparison of the average match scores of the three algorithms under three different requester-service scenarios. And the following section presents the comparison of the average execution time for the same. The parameter settings for the PSI algorithm is set to the default values, such as, number of particles to 25, maximum number of iterations 5000, R_1 and R_2 at 2 and 2 respectively, and the value of W varied from 0.5 to 0.3 over 2000 iterations - for both the experiments.

Comparison of the Average Match Score

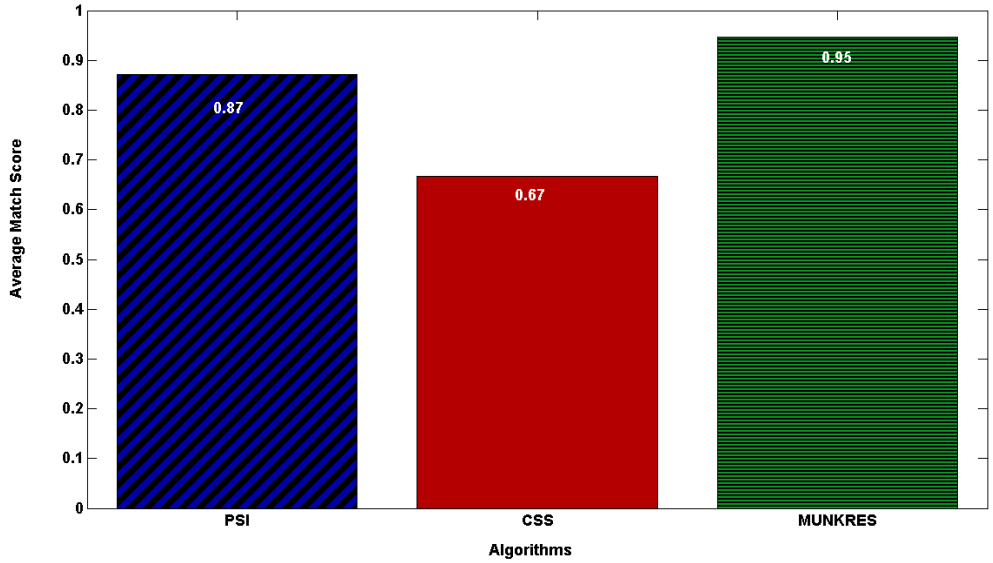

Figure 5.14: Comparison of the average match score for 500-500 requester-service pair.

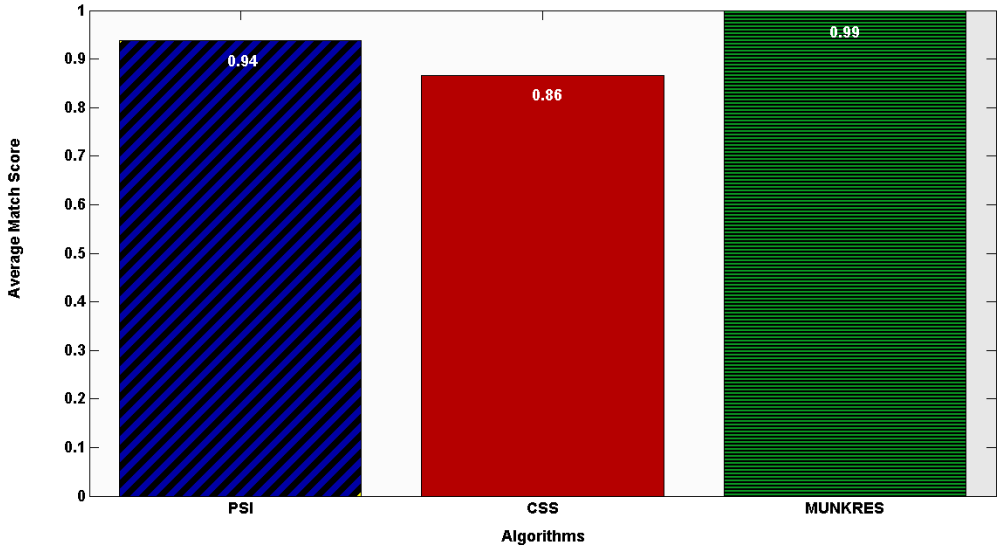

Figure 5.15: Comparison of the average match score for 500-1000 requester-service pair.

The Figures 5.14, 5.15 and 5.16 represents the comparison of the average match score achieved,

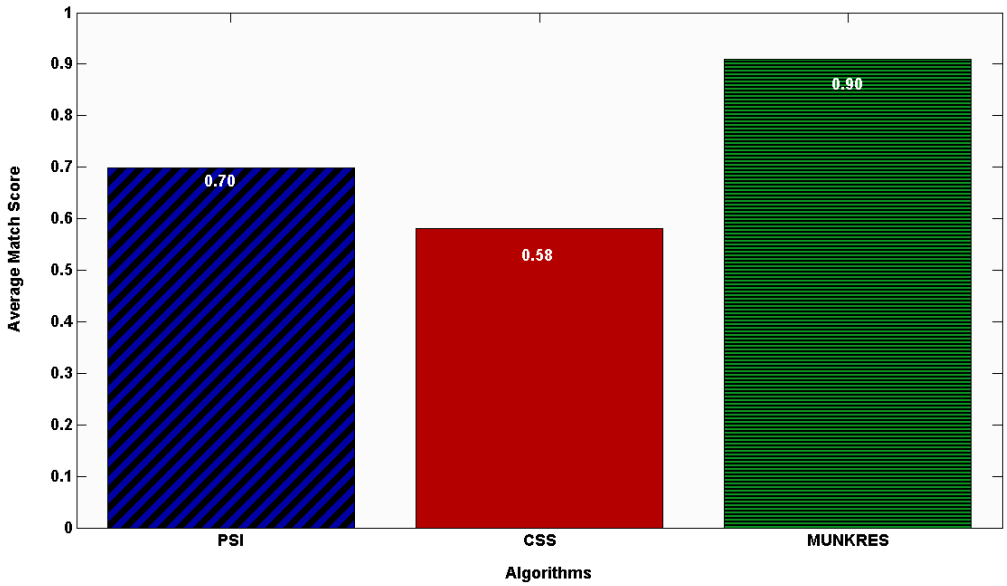

Figure 5.16: Comparison of the average match score for 1000-500 requester-service pair.

by PSI, CSS and Munkres algorithm, for 500-500, 500-1000 and 1000-500 requester service pair respectively. In Figure 5.14, it can be seen that the average match scores achieved by the Munkres algorithm, PSI and CSS are 0.95, 0.87 and 0.67 respectively. The average match score achieved by CSS is approximately 41.78% lower than the Munkres algorithm. Whereas, the PSI algorithm achieves an approximately 9.19% lower match score than Munkres. Also, it can be seen that the average match scores achieved by PSI and Munkres tend more towards the best solution, which is 1, unlike the match score of CSS, which is considerably lower than that. Considering that the Munkres algorithm guarantees an optimal assignment, it can be seen that the PSI algorithm not only achieves a solution closer to that, but also achieves a significantly better result than the CSS algorithm when compared to the Munkres algorithm.

In Figure 5.15, and 5.16, the comparison of average match scores achieved by PSI, CSS and Munkres is presented for 500-1000, and 1000-500 requester service pairs. Although, in these two cases the number of requesters and services are unequal, the number of successful matches is always equal to whichever of the two is less. For example, in this case the number of successful match in both cases is 500. The results in case of 500-1000 shows that the algorithms perform almost comparably to each other by achieving a match score of .94, .86 and .99 for PSI, CSS and Munkres algorithm respectively. In this case, the PSI algorithm achieves a match score which is only 5.3% lower than the match score achieved by the Munkres algorithm. Whereas, CSS achieves a match score almost 15.11% lower than Munkres. Comparing all the graphs, it can be seen that all the algorithms perform best in the case of 500-1000 requester-service pair settings. This can be attributed to the fact that in a scenario where there are more services available for a set of requester, each requester has more options to choose from and hence achieves a better match score.

This scenario also calls for the least optimization, as the competition between the requesters are less.

In 1000-500 requester-service scenario the average match score achieved by the PSI, CSS and Munkres algorithm are 0.70, 0.58 and 0.90 respectively. Here also, Munkres algorithm performs marginally better than PSI and CSS. The percentage difference between the match scores obtained by the PSI versus the Munkres algorithm is 28.57% lower, whereas the match score achieved by the CSS algorithm is 55.17% lower than the Munkres algorithm.

Combining all three scenarios the average match score obtained by Munkres, PSI and CSS is, 0.95, 0.83 and 0.70 respectively. Munkres achieves the best match score. On the other hand, PSI and CSS algorithm achieves a match score which is 14.45% and 35.71% less than Munkres. These results proves that the average match scores achieved by the PSI algorithm is not only much better than the CSS, but is also guarantees a better solution like Munkres algorithm. The reason for higher accuracy of the PSI algorithm can be attributed to the fact that the heuristics implemented by the particles in the swarm, are more powerful than a simple constraint satisfaction based optimization approach, or the Munkres algorithm approach.

The greatest advantage of the PSI algorithm is that it reaches the better solution based on the heuristic method. This is unlike the approach of the Munkres algorithm, in which the optimal solution is reached by considering all the possible solutions to the problem. This nature adds to the complexity and also becomes computationally intensive when there are larger number of requesters and services in the pool. It is also important to note here that, unlike Munkres, because PSI is based on evolutionary strategies it is a non-deterministic algorithm. This means that the solution returned by PSI, may or may not be the best possible solution to the problem. However, a diversification of search, for the global best particle in the swarm introduced by the crowding distance operator, and the varying W of the velocity equation, the proposed PSI algorithm guarantees a solution very close to the optimum under different scenarios.

Comparison of the Average Execution Time

This section discusses the comparison of the average execution time of the algorithms. The Figure 5.17, 5.18, and 5.19 represents the comparison of the average execution time taken, by PSI, CSS and the Munkres algorithm, for 500-500, 500-1000 and 1000-500 requester-service parameter settings respectively.

From Figures 5.17, 5.18 and 5.19 it can be seen that the average execution time for all the scenarios of the Munkres algorithm is the highest at approximately 240sec. Whereas, the average execution times for PSI and CSS algorithm are 26.34sec and 11.91sec respectively. The average execution time of the Munkres algorithm is approximately 10 times and 20 times more than the PSI and the CSS algorithm respectively. The figures show that the proposed algorithms are significantly

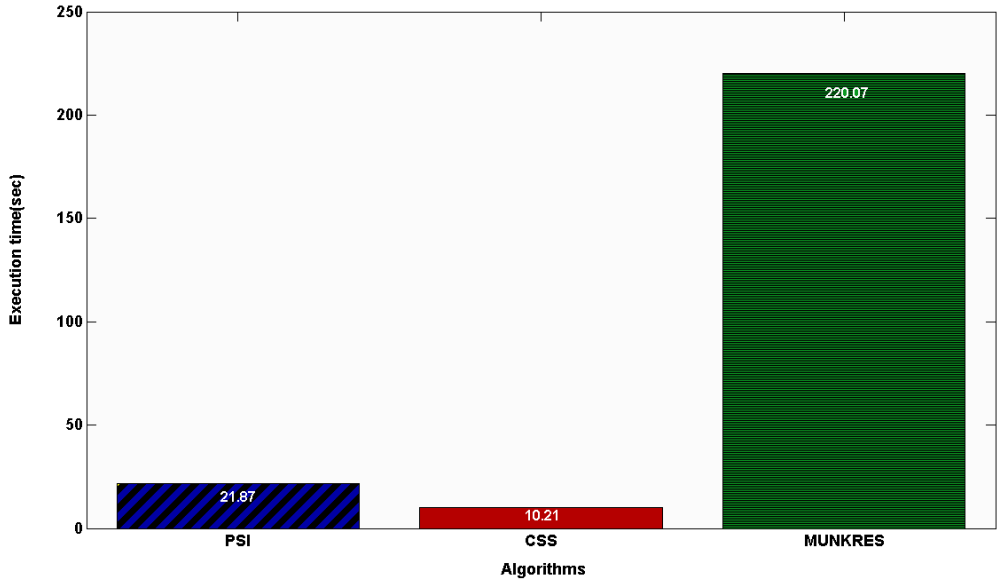

Figure 5.17: Comparison of the execution time(Sec) for 500-500 requester-service pair.

faster than the Munkres algorithm. Although, the CSS algorithm is the fastest, from the previous experimental analysis it shows that it compromises the quality of the match score. Whereas, the Munkres algorithm achieves the best optimal match score out of the three algorithms, but has a significantly high execution time. PSI on the other hand, is a good choice as a service selection algorithm, as it achieves a balance between the accuracy of the match score as well the execution time, thereby proving to be the most efficient compared to the other algorithms.

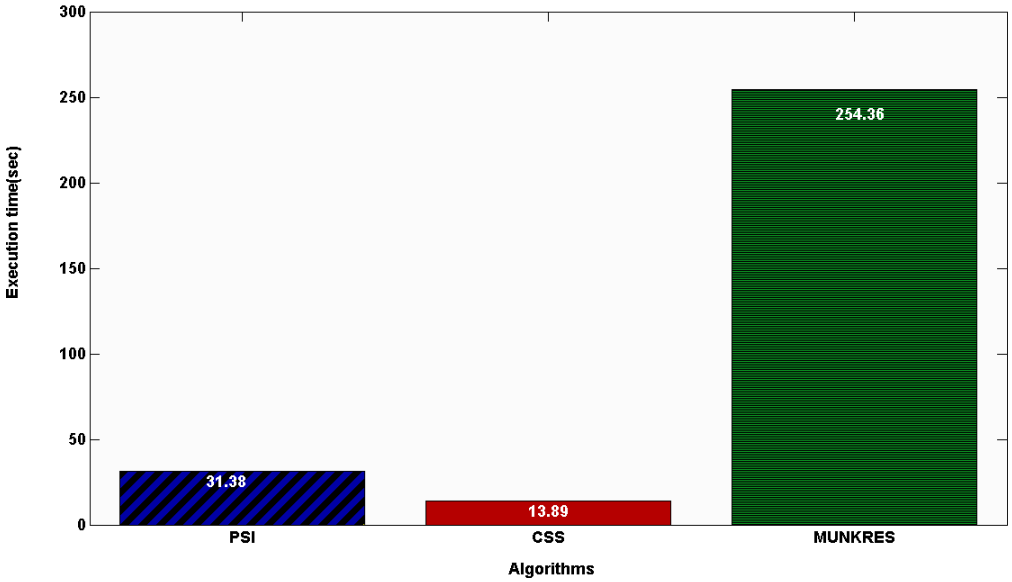

Figure 5.18: Comparison of the execution time(Sec) for 500-1000 requester-service pair.

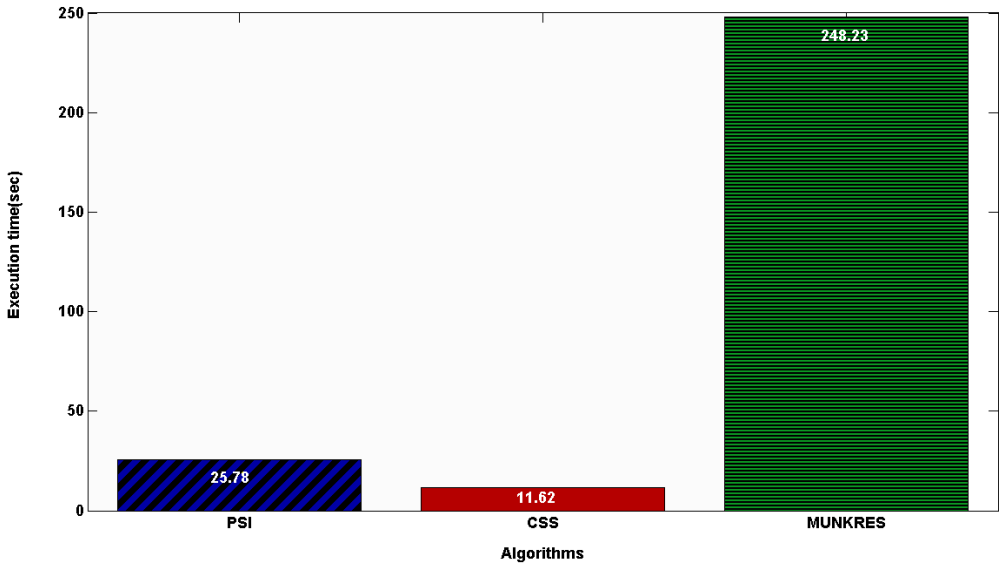

Figure 5.19: Comparison of the execution time(Sec) for 1000-500 requester-service pair.

5.5 Summary

This chapter compared the Multiple Objective Particle Swarm Optimization based service selection (PSI) algorithm to the Constraint Satisfaction based selection (CSS) algorithm. From the results it can be seen that the PSI achieves higher accuracy than the CSS algorithm, when both are compared to the Munkres algorithm. Moreover, looking at the results, it can be concluded that even though the PSI takes relatively longer than the CSS algorithm to compute, it matches the requester-service pairs with a better match score, thereby optimizing the matching process more efficiently. It also proved that using the PSI algorithm for selecting services deployed on the Grid, where several requesters are competing for similar services, achieves a significantly better match score than the CSS algorithm. Furthermore, it can be seen that the PSI algorithm outperformed the CSS algorithm in the scalability analysis. In this case, the algorithms were tested for match score accuracy, for an increasing number of incoming requests against a fixed number of available services. The reason behind the efficiency of the PSI algorithm can be attributed to the three key features of the algorithm:

- Firstly, the algorithm keeps track of the best solutions found in the previous generations. This helps to converge to a better solution with every iteration, because for each iteration the particles compare their current solution to the previously achieved best solution. If the current solution is better than the previously achieved solution it sets it to p_{best}, else keeps the previous obtained best solution in its memory.

- Secondly, the crowding distance operator, helps to diversify the search of the global best

guide, and avoids getting stuck in the local optima of the problem space.

- Thirdly, the inertia weight W of the velocity equation plays a key role in guiding the heuristics to the global best guide. The inertia weight is changed linearly within the first 2000 iterations from 0.5 to 0.3. A higher initial inertia weight guides the global optimum search to the region near the global best guide. A lower inertial weight during the later iterations allows refining the search towards the global optima and hence gives a better solution.

These characteristics of the PSI, along with the experimentation results verifies, that the proposed algorithm, is a powerful and an accurate method to handle multi-objective optimization problems, such as the service selection problem on the Grid.

Chapter 6

Conclusion and Future Work

The Grid is a high performance computing paradigm, which is rapidly emerging as a suitable platform to deploy distributed applications. The feasibility of this type of supercomputing has already been demonstrated by the large number of applications deployed on the Grid. To increase the effectiveness and applicability of these high performance applications on the Grid, having an efficient service selection algorithm has become mandatory. Service selection is therefore considered as one of the basic components of a Grid system, which helps to discover qualified service providers for different Grid users. A service selection algorithm should be efficient to ensure unbiased allocations of requesters to the service providers. To achieve efficiency, the service selection algorithm should match the services and requesters in a way which optimizes the entire allocation process. This thesis has focused on the problem of service selection in Grid systems. The following key issues were addressed:

- Assigning large number of incoming requests to the available services simultaneously;
- Achieving high match scores in the case of a large number of similar types of incoming requests;
- Assigning each requesters accurately, i.e., as closely as possible to all the available services;
- Providing requesters the flexibility to request multiple service selection criteria, based on a QoS metric;
- Selecting the appropriate services for the incoming requests within a reasonable time.

6.1 Summary of Contributions

The contributions made in this thesis primarily include the introduction of two new service selection methods for the Grid services. The findings of the same are summarized as follows:

1. **CSS Algorithm - Constraint satisfaction optimization approach:** Application of constraint satisfaction based optimization approach in the service selection of Grid services, helped to solve the problems that were apparent in the classical matchmaking approach. In

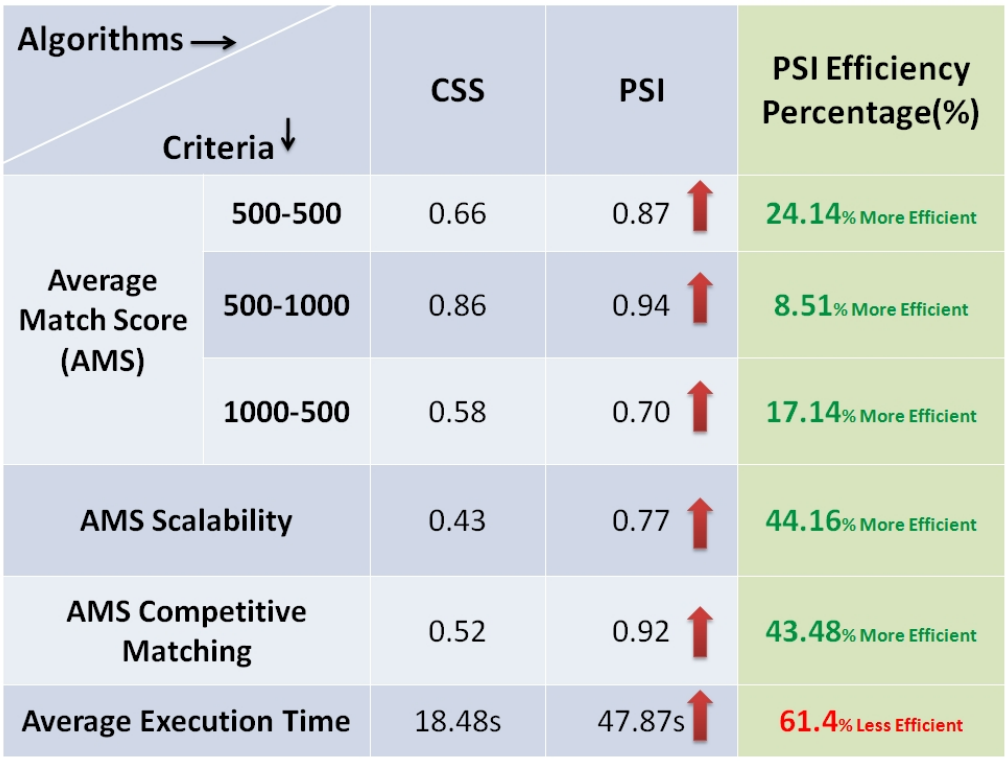

Figure 6.1: Summary of PSI and CSS performance.

particular, this approach helped to overcome the poor assignment of requesters appearing later in the list, which resulted in an overall poor match score. This approach performed the assignment process after considering the match scores produced by all the requesters and the available services in the pools simultaneously. The results found using this approach were very encouraging, when it was compared to the classical service selection algorithm. The average match score achieved in this approach for 500-500, 500-1000 and 1000-500 were 0.66, 0.86 and 0.58 respectively, and the execution time of the algorithm for 1000-1000 requester-service pairs is approximately 46.75s. For 500 services, when the number of requesters were uniformly varied from 500 to 10000, the average match score was approximately 0.43. Furthermore, the average match score and the execution time of this algorithm was 0.52 and 11.48s approximately, when the percentage of similar requests was uniformly varied from 5% to 95% for 500-500 requester-service pairs. The match scores achieved by the CSS algorithm is considerably better than the match score achieved in the traditional service selection algorithm. However, it is still considerably lower than the optimum of 1. An efficient service selection algorithm should be able to match a set of requester-service with a higher and better match score. Hence, this research proposed an service selection algorithm based on swarm intelligence to achieve a better match score. A detailed discussion about the results achieved

by it, and evaluations made are given in the following paragraphs.

2. **PSI Algorithm - A multi-objective particle swarm optimization approach:** This is the primary contribution of the thesis. It introduces the use of natural computing, and more specifically swarm intelligence for service selection on the Grid. The introduction of the particle swarm optimization technique to solve this is rather unique. As it can be seen from the background research, that no prior investigation has been done in this regard from the Grid service selection perspective. The concept is not only new to the Grid service selection process, but considering the results it has also proved to be a better service selection method than the CSS approach. It achieved a considerably better overall match scores for all the evaluated cases. And it also efficiently achieved the main goal of this research, i.e., to accurately assign large number of incoming requests (similar types or otherwise), to the available services with a closer match. The average match score achieved with this approach for 500-500, 500-1000 and 1000-500 are 0.87, 0.94, and 0.70 respectively. The execution time of the algorithm for 1000-1000 requester-service pairs, 5000 iterations, and 25 particles is approximately 132s. The PSI algorithm for 500 services, the average match score was approximately 0.77, when the number of requesters were uniformly varied from 500 to 10000, keeping the particle size, number of iterations fixed to 25 and 5000 respectively. And, also the average match score and the execution time of this algorithm was 0.92 and 36.45s approximately, when the percentage of similar requests was uniformly varied from 5% to 95% for 500-500 requester-service pairs.

Figure 6.1 summarizes the average match score achieved by both algorithms for 500-500, 500-1000, and 1000-500 requester-service pair scenarios. It also represents the average match score achieved while testing the scalability, competitive matching characteristics of the algorithms and the average execution time achieved by the algorithms. The table primarily presents the efficiency percentage of the PSI algorithm over the CSS in terms of the average match score and average execution time achieved in various scenarios. Analyzing the results, it can be concluded that the average match score achieved by the PSI approach is significantly better than the one achieved by the CSS algorithm. Considering all the scenarios, in general the average match score achieved by the PSI algorithm and the CSS algorithm is approximately 0.84 and 0.61. Hence, the overall average match score achieved by the PSI algorithm is 27.38% higher, hence more efficient than the CSS algorithm.

As discussed in the previous chapter, the goal of the service selection problem is to find a service for a requester with a closer match based on the QoS parameter requested by the user. The algorithm seeks to achieve higher overall match scores, by aiming for a closer requester-service QoS parameter match. The PSI algorithm proves itself very efficient in achieving this goal.

On the other hand, the execution time taken by PSI is approximately 61.4% higher than the CSS algorithm. The greater execution time for the PSI algorithm can mainly be attributed to the evolutionary strategies followed by the algorithm. For example, particles keeps a track of the local

best solution and global best solution. And, using this information the particles find the optimized match score between each requester-service pair by iterating the program over certain generations based on the given objective function. It is important note that, although the PSI algorithm does take more time to execute the average match score returned by the algorithm is significantly better than the CSS algorithm.

Algorithms → / Criteria ↓	CSS	PSI	Munkres
Average Fitness Score	0.70	0.84	0.95
Execution Time	11.9s	26.34s	240.88s

Figure 6.2: Comparison of PSI and CSS algorithms with Munkres Assignment Algorithm.

Furthermore, Figure 6.2 represents the results found when the proposed algorithms were compared to the Munkres assignment algorithm. The reason for the comparison is based on the fact that the service selection problem is similar to the assignment problem. And Munkres assignment problem is considered as a standard assignment algorithm in the literature. The Munkres assignment algorithm is also known to guarantee an optimal solution thereby, making it a suitable benchmark to compare the accuracy and the efficiency of the algorithms.

The results proved that the average match score achieved by the PSI algorithm were more accurate and time efficient when compared to CSS and Munkres' algorithm, respectively. The average match score returned by CSS, PSI and Munkres assignment algorithm are 0.7, 0.84 and 0.95 respectively. Hence, it can be seen that the PSI algorithm achieved an average match score, more closer to the optimal solution obtained by the Munkres algorithm. PSI achieved an average match score which is only 13.09% lower than the Munkres assignment algorithm. Whereas, CSS achieves a match score which is 35.71% and 20% less optimal than the Munkres and the PSI algorithm respectively. Thereby, PSI accomplished a better match score out of the three algorithms. Clearly CSS achieves the worse match score out of the three algorithm. The PSI achieves a higher match score because it avoids the biased matches, and selects one of the non-dominated global best solutions as the best solution.

On the other hand, from Figure 6.2 can be seen that, the execution times of the three algorithms reveals that the CSS algorithm is the fastest. The average execution time achieved by the three

algorithms is 11.9s, 26.34s and 240.88s for CSS, PSI and Munkres algorithm respectively. From the results, it can be inferred that, although PSI has a marginally higher average execution time than CSS algorithm, the average execution time of the PSI algorithm is significantly smaller than the Munkres algorithm. PSI is approximately 9 times faster than the Munkres algorithm. Hence, the experimental results of the proposed approaches shows that PSI is recommended as an efficient service selection approach for the Grid services, as it balances the execution time and the accuracy during the matching process.

Furthermore, the most important strength of swarm intelligence based PSI algorithm is that the search for the global best guide, or a better solution is based on the heuristic method. This method proves very efficient, both accuracy and execution time wise, when there are large numbers of requesters and services, as in the case of the Grid scenario. On the other hand, the Munkres algorithm relies more on searching all the possible solutions before coming up with the best one. This method proves inefficient, execution time wise when there are too many requester-services accessing the Grid at the same time.

6.2 Future Work

The key of using swarm intelligence based techniques in the service selection process, lies in the fact that they are different from traditional search algorithms, as it usually works on a population of potential solutions of the problem. It usually uses a heuristic to solve complex combinatorial optimization problems. Thereby, using an informal way of reaching to a better solution with each iteration, instead of actually going through all the possible best solutions, which can be time consuming. When applied to complex optimization problems, through cooperation and competition among the potential solutions, this technique can find optima more efficiently and often quicker than genetic algorithms. This research proposed the application of the particle swarm optimization method to solve the service selection problem. The proposed PSI algorithm introduced the concept of using swarm intelligence based methods for selecting Grid services. As with any introductory concept, this technique also has room for more improvement. A few possible future improvements of this research is discussed in this section as follows:

- **Include composite work flow concept:** The swarm intelligence based selection algorithm may be extended to see its feasibility when applied to the selection process that entails the selection of more than one service to completely execute the client request. The selection of these services are usually defined as a workflow. It primarily defines the sequence of the services to be selected to achieve a successful execution of the job. Integrating the composite service selection process with the existing service selection algorithm, will help to assign requests that require multiple services, and will further increase the scope of the algorithm.

- **Extension to Web Services:** The Web Service technology, just like the Grid, is a rapidly growing area in the field of computer science. Both share some common core concepts, one of them being the selection of the Web Services for a job submitted by the users. The proposed algorithm can be implemented to select the Web Services in a similar manner as we saw for the Grid.

- **Artificial Neural Network based training concept:** Artificial Neural Network concept, can be used to train the particles at first with some prototype test cases. This would make them more efficient to solve the objective function of the given problem. This may help to achieve better optimization results, although it may compromise the execution time of the algorithms simultaneously.

Summarizing, the primary contribution of this thesis is the introduction of Swarm Intelligence based approach to the Grid service selection process. This thesis proposed a novel swarm intelligence based approach - the PSI algorithm - for optimizing the service selection process on the Grid. It also proposed a second algorithm - the CSS algorithm - as an enhancement of the traditional service selection algorithm. The experimental analysis and evaluations done, prove that the PSI algorithm performs significantly better than the CSS algorithm, hence making it a better selection method for the Grid Services.

References

[1] S.A. Ludwig and S. M. S. Reyhani. Selection algorithm for grid services based on a quality of service metric. In *HPCS '07: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, page 13, Washington, DC, USA, 2007. IEEE Computer Society.

[2] J.B. Weissman. Adaptive Resource Selection for Grid-Enabled Network Services. *Proceedings of the Second IEEE International Symposium on Network Computing and Applications*, 2003.

[3] M. Baker, R. Buyya, and D. Laforenza. Grids and Grid technologies for wide-area distributed computing. *Software-Practice and Experience*, 32(15):1437–66, 2002.

[4] I. Foster. What is the Grid? A Three Point Checklist. *Grid Today*, 1(6):22–25, 2002.

[5] I.T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 1–4, London, UK, 2001. Springer-Verlag.

[6] R. Sobie, A. Agarwal, and P. Armstrong. Uvic grid research, www.grid.phys.uvic.ca/assets/sunfigures/grid-2.jpg.

[7] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[8] R. Alfieri, R. Cecchini, V. Ciaschini, L. Dell'Agnello, A. Frohner, A. Gianoli, K. Lorentey, and F. Spataro. VOMS, an authorization system for virtual organizations. *Lecture notes in computer science*, pages 33–40.

[9] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer*, pages 37–46, 2002.

[10] J. Anselmi, D. Ardagna, and P. Cremonesi. A qos-based selection approach of autonomic grid services. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 1–8, New York, NY, USA, 2007. ACM.

[11] Y. Liu and H. He. Grid service selection using qos model. In *SKG '07: Proceedings of the Third International Conference on Semantics, Knowledge and Grid*, pages 576–577, Washington, DC, USA, 2007. IEEE Computer Society.

[12] R. Al-Ali, O. Rana, D. Walker, S. Jha, and S. Sohail. G-QoSM: Grid Service Discovery Using QoS Properties. *Special Issue on Grid Computing and Informatics Journal*, 21(4):363–382, 2002.

[13] S. Andreozzi. On the quality-based evaluation and selection of grid services. *Technical Report UBLCS-2006-2, Department of Computer Science, University of Bologna*, 2006.

[14] A. Four. International science grid this week: Feature - mission: Healthgrid,. HealthGrid, http://www.isgtw.org/images/humangridinitiative.jpg.

[15] K. Dean and T. Solomonides. HealthGrid-a Summary. Cisco White Paper.

[16] I. Blanquer, V. Hernandez, et al. Healthgrid–a summary, a joint paper from the Healthgrid Association and Cisco Systems, 2004.

[17] V. Breton, K. Dean, T. Solomonides, et al. The Healthgrid White Paper. *Healthgrid*, pages 249–318, 2005.

[18] S.K. Das and N. Deo. Parallel hungarian algorithm. *Computer Systems Science and Engineering*, 5(3):131–136, 1990.

[19] H. Feltl and G.R. Raidl. An improved hybrid genetic algorithm for the generalized assignment problem. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 990–995, New York, NY, USA, 2004. ACM.

[20] R. Silver. An algorithm for the assignment problem. *Commun. ACM*, 3(11):605–606, 1960.

[21] F. Bourgeois and J.C. Lassalle. An extension of the munkres algorithm for the assignment problem to rectangular matrices. *Commun. ACM*, 14(12):802–804, 1971.

[22] M. Bichler and K.J. Lin. Service-Oriented Computing. *Computer*, pages 99–101, 2006.

[23] S. Cuddy, M. Katchabaw, and H. Lutfiyya. Context-aware service selection based on dynamic and static service attributes. *IEEE International Conference on Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005)*, 4:13–20 Vol. 4, Aug. 2005.

[24] X. Zhang, J.L. Freschl, and J.M. Schopf. A performance study of monitoring and information services for distributed systems. *12th IEEE International Symposium on High Performance Distributed Computing, 2003. Proceedings.*, pages 270–281, 2003.

[25] S.A. Ludwig and S.M.S. Reyhani. Semantic approach to service discovery in a Grid environment. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(1):1–13, 2006.

[26] F. Dong and S.G. Akl. Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. *Queen's University School of Computing. January*, 2006.

[27] C. Liangyin, L. Zhishu, L. Qing, Z. Jingyu, C. Yanhong, and C. Liangwei. An Approach toDynamic Grid Service Selection Based on Improved Reinforcement Q-learning. *The First International Symposium on Data, Privacy, and E-Commerce, 2007. ISDPE 2007.*, pages 412–414, 2007.

[28] E. Elmroth and J. Tordsson. Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions. *Future Generation Computer Systems*, 24(6):585–593, 2008.

[29] R. Buyya, D. Abramson, and J. Giddy. A Case for Economy Grid Architecture for Service-Oriented Grid Computing. *10th IEEE International Heterogeneous Computing Workshop (HCW 2001), In conjunction with IPDPS*, 2001.

[30] B. Alunkal, I. Veljkovic, G. von Laszewski, and K. Amin. Reputation-Based Grid Resource Selection. *Proceedings of AGridM*, 2003.

[31] Ziqiang Xu, P. Martin, W. Powley, and F. Zulkernine. Reputation-enhanced qos-based web services discovery. *IEEE International Conference on Web Services, 2007. ICWS 2007.*, pages 249–256, July 2007.

[32] R. Buyya and M. Murshed. A Deadline and Budget Constrained Cost-Time Optimisation Algorithm for Scheduling Task Farming Applications on Global Grids. *Arxiv preprint cs/0203020*, 2002.

[33] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of Ninf: towards a global computing infrastructure. *Future Generation Computer Systems*, 15(5-6):649–658, 1999.

[34] F. Klan. Context-aware service discovery, selection and usage. *Grundlagen von Datenbanken*, pages 95–99, 2006.

[35] K. Lee and M. Kim. Service selection for ubiquitous computing environment using situation awareness. *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006.*, pages 5 pp.–, April 2006.

[36] G. Fenza, V. Loia, and S. Senatore. Improving fuzzy service matchmaking through concept matching discovery. *IEEE International on Fuzzy Systems Conference, 2007. FUZZ-IEEE 2007.*, pages 1–6, July 2007.

[37] I. Mecar, A. Devlic, and K. Trzec. Agent-oriented semantic discovery and mathcmaking of web services. *Proceedings of the 8th International Conference on Telecommunications, 2005. ConTEL 2005.*, 2:603–607, June 2005.

[38] C.Q. Huang, D.R. Chen, and H.L. Hu. Intelligent agent-based scheduling mechanism for grid service. *Proceedings of 2004 International Conference on Machine Learning and Cybernetics, 2004.*, 1, 2004.

[39] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 63, Washington, DC, USA, 2002. IEEE Computer Society.

[40] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988.

[41] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. *Grid Computing: Making The Global Infrastructure a Reality, John Wiley*, pages 0–470, 2003.

[42] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[43] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughputcomputing. In *The Seventh International Symposium on High Performance Distributed Computing, 1998. Proceedings.*, pages 140–146, 1998.

[44] R. Raman. *Matchmaking Frameworks for Distributed Resource Management.* PhD thesis, University of Wisconsin, 2001.

[45] J.M. Schopf. A General Architecture for Scheduling on the Grid. *Submitted to special issue of Journal of Parallel and Distributed Computing Systems on Grid Computing*, 2002.

[46] S.A. Ludwig and S. M. S. Reyhani. Introduction of semantic matchmaking to grid computing. *J. Parallel Distrib. Comput.*, 65(12):1533–1541, 2005.

[47] X. Bai, H. Yu, Y. Ji, and D.C. Marinescu. Resource matching and a matchmaking service for an intelligent grid. *International Journal of Computational Intelligence*, 1(3):197–205, 2004.

[48] A. Harth, S. Decker, Y. He, H. Tangmunarunkit, and C. Kesselman. A semantic matchmaker service on the grid. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 326–327, 2004.

[49] H. Zhuge. Semantics, Resource and Grid. *Future Generation Computer Systems*, 20(1):1–5, 2004.

[50] S. Ludwig and P.V. Santen. A grid service discovery matchmaker based on ontology description, 2002.

[51] N.V. Neela and S. Kailash. Resource matchmaking in grid - semantically. *The 9th International Conference on Advanced Communication Technology*, 3:2051–2055, Feb. 2007.

[52] S. Ran. A model for web services discovery with QoS. *ACM SIGecom Exchanges*, 4(1):1–10, 2003.

[53] Y. Liu. QoS Computation and Policing in Dynamic Web Service Selection, WWW2004, New York. *IEEE computer Society*, pages 66–73, 2004.

[54] E.M. Maximilien and M.P. Singh. A framework and ontology for dynamic Web services selection. *IEEE Internet Computing*, 8(5):84–93, 2004.

[55] E.M. Maximilien and M.P. Singh. Toward autonomic web services trust and selection. *Proceedings of the 2nd international conference on Service oriented computing*, pages 212–221, 2004.

[56] M. Ruse. Darwin and His Critics: The Reception of Darwin's Theory of Evolution by the Scientific Community. *Philosophy of Science*, 42(3):338–339, 1975.

[57] M. Mitchell. *An Introduction to Genetic Algorithms*. Bradford Books, 1996.

[58] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858, 2000.

[59] I. Foster. The Grid: A New Infrastructure for 21st Century Science. *Physics Today*, 55(2):42–47, 2002.

[60] C. Darwin. *Charles Darwin's Natural Selection: Being the Second Part of His Big Species Book Written from 1856 to 1858*. Cambridge University Press, 1987.

[61] C. Darwin and J. Carroll. *On the Origin of Species*. Broadview Press, 2003.

[62] O. Hrstka, A. Kučerová, M. Lepš, and J. Zeman. A competitive comparison of different types of evolutionary algorithms. *Computers and Structures*, 81(18-19):1979–1990, 2003.

[63] D. Whitley, S. Rana, J. Dzubera, and K.E. Mathias. Evaluating evolutionary algorithms. *Artificial Intelligence*, 85(1-2):245–276, 1996.

[64] D. Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.

[65] J.L. Henderson. Genetic algorithm flowchart. http://www.sv.vt.edu/.../henderson/genalgo.gif, October 22 2000. Department of Aerospace and Ocean Engineering, Virginia Tech.

[66] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[67] J.D. Schaffer. Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, 1985.

[68] J. Horn, N. Nafpliotis, and D.E. Goldberg. A niched Pareto genetic algorithm for multiobjective optimization. *Proceedings of the First IEEE Conference on Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence.*, pages 82–87, 1994.

[69] J. Kennedy and R. Eberhart. Particle swarm optimization. *IEEE International Conference on Neural Networks, 1995. Proceedings.*, 4, 1995.

[70] Y. Shi and R.C. Eberhart. Empirical study of particle swarm optimization. *Proceedings of the 1999 Congress on Evolutionary Computation, 1999. CEC 99.*, 3:–1950 Vol. 3, 1999.

[71] J. Robinson, S. Sinton, and Y. Rahmat-Samii. Particle swarm, genetic algorithm, and their hybrids: optimization of a profiled corrugated horn antenna. *Antennas and Propagation Society International Symposium, 2002. IEEE*, 1:314–317 vol.1, 2002.

[72] J. Kennedy and R.C. Eberhart. Swarm intelligence. *Morgan Kaufmann Publishers Inc. San Francisco, CA, USA*, page 512, 2001.

[73] R.C. Eberhart and Y. Shi. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 81–86. Piscataway, NJ, USA: IEEE, 2001.

[74] G. Coath and S.K. Halgamuge. A comparison of constraint-handling methods for the application of particle swarm optimization to constrained nonlinear optimization problems. In *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, volume 4, 2003.

[75] S. Doctor, GK Venayagamoorthy, and VG Gudise. Optimal PSO for collective robotic search applications. In *Congress on Evolutionary Computation, 2004. CEC2004.*, volume 2, 2004.

[76] E.A. Grimaldi, F. Grimaccia, M. Mussetta, R.E. Zich, and D. di Elettrotecnica. PSO as an effective learning algorithm for neural network applications. pages 557–560, 2004.

[77] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001.

[78] E. Zitzler. Evolutionary Algorithms for Multiobjective Optimization. *Methods and Applications PhD thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, November*, 1999.

[79] R.M. Everson, J.E. Fieldsend, and S. Singh. Full Elite Sets for Multi-objective Optimisation. *Adaptive Computing in Design and Manufacture V*, 2002.

[80] P. Korhonen and M. Halme. Using Lexicographic Parametric Programming for Searching a Non-dominated Set in Multiple-Objective Linear Programming. *Journal of Multi-Criteria Decision Analysis*, 5(4):291–300, 1996.

[81] J.D. Knowles and D.W. Corne. Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

[82] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195, 2000.

[83] C.A.C. Coello and M.S. Lechuga. MOPSO: A Proposal for Multiple Objective Particle Swarm Optimization. *Congress on Evolutionary Computation (CEC2002)*, 2:1051–1056.

[84] J.E. Fieldsend, R.M. Everson, and S. Singh. Using unconstrained elite archives for multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 7(3):305–323, 2003.

[85] X. Hu and R. Eberhart. Multiobjective optimization using dynamic neighborhood particleswarm optimization. *Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC'02.*, 2, 2002.

[86] K.E. Parsopoulos and M.N. Vrahatis. Particle swarm optimization method in multiobjective problems. *Proceedings of the 2002 ACM symposium on Applied computing*, pages 603–607, 2002.

[87] J. Fieldsend and S. Singh. A multi-objective algorithm based upon particle swarm optimisation. *Proceedings of The UK Workshop on Computational Intelligence*, pages 34–44, 2002.

[88] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. *10th IEEE International Symposium on High Performance Distributed Computing*, 184, 2001.

[89] S. Fitzgerald, I. Foster, C. Kesselman, G.V. Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. *Proc. 6th IEEE Symposium on High Performance Distributed Computing*, 375, 1997.

[90] C.R. Raquel and P.C. Naval. An effective use of crowding distance in multiobjective particle swarm optimization. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 257–264, New York, NY, USA, 2005. ACM.

[91] R. Xiao, B. Li, and X. He. The particle swarm: Parameter selection and convergence. In *Advanced Intelligent Computing Theories and Applications. With Aspects of Contemporary Intelligent Computing Techniques*, volume Volume 2 of *Communications in Computer and Information Science*, pages 396–402, Qingdao, China, August, 2007 2007. Springer Berlin Heidelberg.

[92] I.C. Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information Processing Letters*, 85(6):317–325, 2003.

[93] B. Birge. PSOt-a particle swarm optimization toolbox for use with Matlab. In *Swarm Intelligence Symposium, 2003. SIS'03. Proceedings of the 2003 IEEE*, pages 182–186, 2003.

[94] D.B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1):461–474, 1993.

[95] P.C. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *Journal of the Society for Industrial and Applied Mathematics*, 10(2):305–313, 1962.

[96] M. Savelsbergh. A Branch-and-Price Algorithm for the Generalized Assignment Problem. *Operations Research Baltimore-*, 45:831–841, 1997.

[97] P.C. Chu and J.E. Beasley. A genetic algorithm for the generalised assignment problem. *Computers and Operations Research*, 24(1):17–23, 1997.

[98] G.T. Ross and R.M. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical Programming*, 8(1):91–103, 1975.

[99] T. Stutzle and M. Dorigo. ACO algorithms for the quadratic assignment problem. *New Ideas in Optimization*, pages 33–50, 1999.

[100] H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics*, 52(1):7–21, 2005.

[101] H.W. Kuhn. The hungarian method for solving the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[102] J. Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5:32, 1957.

[103] P.J Angeline. Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences. In *EP '98: Proceedings of the 7th International Conference on Evolutionary Programming VII*, pages 601–610, London, UK, 1998. Springer-Verlag.

[104] A. Augugliaro, L. Dusonchet, and E.R. Sanseverino. Evolving non-dominated solutions in multiobjective service restoration for automated distribution networks electric power systems research 59.

[105] U. Baumgartner, Ch. Magele, and W. Renhart. Pareto optimality and particle swarm optimization. *Magnetics, IEEE Transactions on*, 40(2):1172–1175, March 2004.

[106] C.A.C. Coello, G.T. Pulido, and M.S. Lechuga. Handling multiple objectives with particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):256–279, June 2004.

[107] R.C. Eberhart and Y. Shi. Comparison between genetic algorithms and particle swarm optimization. In *EP '98: Proceedings of the 7th International Conference on Evolutionary Programming VII*, pages 611–616, London, UK, 1998. Springer-Verlag.

[108] Y. Guangyou. A modified particle swarm optimizer algorithm. *8th International Conference on Electronic Measurement and Instruments, 2007. ICEMI '07.*, pages 2–675–2–679, 16 2007-July 18 2007.

[109] A. Hossain, M. Shamim, A. Atif, and E.S. Abdulmotaleb. Qos-aware service selection for multimedia transcoding. *IMTC 2008. IEEE Instrumentation and Measurement Technology Conference Proceedings, 2008.*, pages 588–593, May 2008.

[110] Z. Li-ping, Y. Huan-jun, and H. Shang-xu. Optimal choice of parameters for particle swarm optimization. In Peter H. Beyers Wei yang, editor, *Journal of Zhejiang University SCIENCE*, volume 6A of *JZUS-A*, pages 528–534, Hongzhou, China, 2005. Zhejang University, Zhejiang University Press.

[111] R.A. Pilgrim. Munkres' assignment algorithm, modified for rectangular matrices. Link.

[112] S.T. Selvi, R.A. Balachandar, K. Vijayakumar, N. Mohanram, M. Vandana, and R. Raman. Semantic discovery of grid services using functionality based matchmaking algorithm. *IEEE/WIC/ACM International Conference on Web Intelligence, 2006. WI 2006.*, pages 170–173, Dec. 2006.

[113] Y. Shi and R.C. Eberhart. Fuzzy adaptive particle swarm optimization. *Proceedings of the 2001 Congress on Evolutionary Computation, 2001.*, 1:101–106 vol. 1, 2001.

[114] Y. Shi and R.C. Eberhart. Parameter selection in particle swarm optimization. In *EP '98: Proceedings of the 7th International Conference on Evolutionary Programming VII*, pages 591–600, London, UK, 1998. Springer-Verlag.

[115] Y. Shi and R. Eberhart. A modified particle swarm optimizer. *The 1998 IEEE International Conference on Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence.,*, pages 69–73, May 1998.

[116] P.N. Suganthan. Particle swarm optimiser with neighbourhood operator. *Proceedings of the 1999 Congress on Evolutionary Computation, 1999. CEC 99.*, 3:–1962 Vol. 3, 1999.

[117] L.H. Vu, M. Hauswirth, and K. Aberer. QoS-based service selection and ranking with trust and reputation management. *Proceedings of the International Conference on Cooperative Information Systems (CoopIS 2005)*, 2005.

[118] F. Wang and Y. Qiu. A modified particle swarm optimizer with roulette selection operator. *Proceedings of 2005 IEEE International Conference on Natural Language Processing and Knowledge Engineering, 2005. IEEE NLP-KE '05.*, pages 765–768, Oct.-1 Nov. 2005.

[119] T. Yu and K.J. Lin. Service selection algorithms for web services with end-to-end qos constraints. *IEEE International Conference on e-Commerce Technology, 2004. CEC 2004. Proceedings.*, pages 129–136, July 2004.

[120] C. Zhihua, Z. Jianchao, and C. Xingjuan. A new stochastic particle swarm optimizer. *Congress on Evolutionary Computation, 2004. CEC2004.*, 1:316–319 Vol.1, June 2004.

Appendix A

Source Code

A.1 Main Functions:

This section presents the code for the main functions of the program that calls the three algorithms sequentially.

```
% main function that requires to be run.
%
% prompts the user to input the number of requesters and also the number of
% services. and also the number of time you want to run the programs.
%
% Finally takes the average of the time, and the match score returned by
% all the three algorithm and plots them in a graph.
%
%Created By: Tapashree Guha, March 2009.
function mainRun
clc
clear
avg_ms=fopen('avg_match.txt','a');
avg_exec=fopen('avg_exec.txt','a');

r=input('Please insert the number of requesters:');
s=input('Please insert the number of services:');
n=input('Please enter the number of times you want to run the program:');
ans_s=zeros(n,4);
for i=1:n
     [t1,m1,t2,m2]=run_psi_css(r,s);
     ans_s(i,1)= t1;
     ans_s(i,2)= m1;
     ans_s(i,3)= t2;
     ans_s(i,4)= m2;
end

T_pso=sum(ans_s(:,1));
M_pso=sum(ans_s(:,2));
T_mm=sum(ans_s(:,3));
M_mm=sum(ans_s(:,4));

avg_T_pso=T_pso/n;
avg_M_pso=M_pso/n;
avg_T_mm=T_mm/n;
avg_M_mm=M_mm/n;
fprintf(avg_ms,'%f %f\n',avg_M_pso,avg_M_mm);
fprintf(avg_exec,'%f %f\n',avg_T_mm, avg_T_pso);

[t3,M3]=munkres(r,s);
T_munkres=t3;
M_munkres=1-M3;
t=[avg_T_pso;avg_T_mm;t3];
MS=[avg_M_pso;avg_M_mm;M_munkres];

%Plot the time
figure(2)
h1=bar(t,'group');
ch1 = get(h1,'Children');
fvd1 = get(ch1,'Faces');
fvcd1 = get(ch1,'FaceVertexCData');
[zs1, izs1] = sortrows(t,1);
for i1 = 1:3
    row1 = izs1(i1);
```

```
    fvcd1(fvd1(row1,:)) = i1;
end
set(ch1,'FaceVertexCData',fvcd1)
set(gca,'XTick',1:1:3)
set(gca,'XTickLabel',{'PSI','CSS','MUNKRES'})
xlabel('Algorithms')
ylabel('Execution time(ms)')
title 'Execution Times'

%plot the Match Scores
figure(3)
h=bar(MS,'group');
ch = get(h,'Children');
fvd = get(ch,'Faces');
fvcd = get(ch,'FaceVertexCData');
[zs, izs] = sortrows(MS,1);
for i = 1:3
    row = izs(i);
    fvcd(fvd(row,:)) = i;
end
set(ch,'FaceVertexCData',fvcd)
set(gca,'XTick',1:1:3)
set(gca,'XTickLabel',{'PSI','CSS','MUNKRES'})
xlabel('Algorithms')
ylabel('Average Match Score')
title 'Average MatchScore Values'

fclose(avg_ms);
fclose(avg_exec);
```

```
% A function to run PSI and CSS Algo at the same time. Called by the
% function mainRun.m
%
% Created By: Tapashree Guha, April 2009.
function [t1,M1,t2,M2]=run_psi_css(r_r,s_s)
fid_inp=fopen('input.txt','wt');
[t1,M1]=psi_def(r_r,s_s);
[t2,M2]=go_cal_css(r_r,s_s);
```

A.2 PSI Algorithm Code

This section gives the source code for the PSI algorithm. [1]

```
% psi_def.m
%
% A function that defines the Input parameters, such as the requesters,
% services for the user. Also fixes the % PSO input parameters, for the
% go_cal_MOPSO function. And finally returns % the execution time e,
% and final average Match Score.
%
% Created by Tapashree Guha: March 2009.


function [e,MS]=psi_def(req,srv)
format long
global A B qosAtt rangeP
if(nargin<1)
    error('Not Enough input arguments');
else
    t=cputime;
    %Initializing Variables
    x=req; y=srv;
    S=rand(y,5); R=rand(x,5);
    A=rand(req,5); %Requesters
    B=rand(srv,5); %Services

    % writing the Service — Requester QoS Matric to Files, also the
    % Matchscore to a file.
    fid = fopen('service.txt', 'wt');
    fid2 = fopen('requester.txt', 'wt');
    fprintf(fid2, '%f  %f  %f  %f %f\n', A);
    fprintf(fid, '%f  %f  %f  %f  %f\n', B);
    fclose(fid);
    fclose(fid2);

    if req<srv
        R=A;
        S=getSrv(req,srv);
        qosAtt=[R S];

    else if req>srv
            R=getReq(req,srv);
            S=B;
            qosAtt=[R S];

        else if req==srv
                R=A;
                S=B;
                qosAtt=[R S];

            end
        end
    end
    %getting dimenstions of A and B
    a=length(R(:,1));
    b=length(S(:,1));


    P=fix(1+(10-1)*rand(a,2));

    rangeP=sort(P,2);
    for i=1:a
```

[1] I, would like to thank [93] for his fine contribution of the MATLAB PSOtoolbox, which has been modified and used, to solve the problem in the thesis. I would also like to thank [90], for introducing crowding distance technique to the PSO, which has provided a lot of inspiration in this work.

```
        if (rangeP(i,1)==rangeP(i,2)&& rangeP(i,1)≠1)
            rangeP(i,1)=rangeP(i,1)−1;
        else if (rangeP(i,1)==rangeP(i,2) && rangeP(i,1)==1)
                rangeP(i,2)=rangeP(i,2)+1;
            end
        end
    end

    dim=length(qosAtt(:,1)); %dimension
    Pdef = [50 5000 25 2 2 0.5 0.3 1500 1e−25 150 NaN 2 0]; %PSO Parameters
    [OUT]=go_cal_psi('obj_cal_psi',dim,1,rangeP,1,Pdef); % Global Bests
    out=abs(OUT);
    inP=out(1:dim);

    [F]=obj_cal_psi(inP');
    MS=1−(F/10);
    e=cputime−t;

end
```

```
% obj_cal_mopso.m
%
% Calculates the objective function for the go_cal_MOSPO function. This
% means that it basically calculates the match score for each R_S pair.
% Returs the value of the objective function after evaluting the objective
% function equation as proposed in the thesis.
%
% Created By: Tapashree Guha, March 2009.

function [F]=obj_cal_psi(X)
global A B qosAtt rangeP %#ok<NUSED>
n=length(qosAtt(:,1));
[r c]=size(X); %#ok<NASGU>
W=X(1:r,1:n);
[r_w c_w]=size(W);
max_w=max(max(W));
for v=1:r_w
    for u=1:c_w
        w(v,u)=W(v,u)/max_w;
    end
end


Ra1=qosAtt(:,1);
Rb1=qosAtt(:,2);
Rc1=qosAtt(:,3);
Rd1=qosAtt(:,4);
Re1=qosAtt(:,5);
Sa2=qosAtt(:,6);
Sb2=qosAtt(:,7);
Sc2=qosAtt(:,8);
Sd2=qosAtt(:,9);
Se2=qosAtt(:,10);

%Calculating the Match Value for each R−S pairs
a1=abs((Ra1−Sa2)./Ra1);
b1=abs((Rb1−Sb2)./Rb1);
c1=abs((Rc1−Sc2)./Rc1);
d1=abs((Rd1−Sd2)./Rd1);
e1=abs((Re1−Se2)./Re1);

%Normalizing them by dividing them with the highest number in the array
A_max=max(a1);
A=a1./A_max;

B_max=max(b1);
B=b1./B_max;

```

```
C_max=max(c1);
C=c1./C_max;

D_max=max(d1);
D=d1./D_max;

E_max=max(e1);
E=e1./E_max;
E=sum(E);

F=abs(1-abs(w*(A+B+C+D+E)/(5*n)));
```

```
% go_cal_psi.m
% Usage:
%  [optOUT]=  PSO(functname,D,mv,VarRange,minmax,PSOparams,plotfcn)
%
% Inputs:
%    functname - string of matlab function to optimize
%    D - # of inputs to the function (dimension of problem)
%
% Optional Inputs:
%    mv - max particle velocity, either a scalar or a vector of length D
%           (this allows each component to have it's own max velocity),
%           default = 4, set if not input or input as NaN
%
%    VarRange - matrix of ranges for each input variable,
%    minmax = 0, funct minimized (default)
%           = 1, funct maximized
%
%    PSOparams - PSO parameters
%      P(1) - Epochs between updating display, default = 100. if 0,
%             no display
%      P(2) - Maximum number of iterations (epochs) to train, default = 2000.
%      P(3) - population size, default = 24
%
%      P(4) - acceleration const 1 (local best influence), default = 2
%      P(5) - acceleration const 2 (global best influence), default = 2
%      P(6) - Initial inertia weight, default = 0.9
%      P(7) - Final inertia weight, default = 0.4
%      P(8) - Epoch when inertial weight at final value, default = 1500
%      P(9)- minimum global error gradient,
%                 if abs(Gbest(i+1)-Gbest(i)) < gradient over
%                 certain length of epochs, terminate run, default = 1e-25
%      P(10)- epochs before error gradient criterion terminates run,
%                 default = 150, if the SSE does not change over 250 epochs
%                                then exit
%      P(11)- error goal, if NaN then unconstrained min or max, default=NaN
%      P(12)- type flag (which kind of PSO to use)
%                 0 = Common PSO w/intertia (default)
%                 1 = Trelea types 1
%                 2 =  MOPSO
%      P(13)- PSOseed, default=0
%                = 0 for initial positions all random
%
%
%    plotfcn - optional name of plotting function, default 'goplotpso',
%              make your own and put here
%
%    PSOseedValue - initial particle position, depends on P(13), If
%    P(13)=0, it means the initial position of the particle is all randomly
%    generated depending on the Varrange
%
% Brian Birge
% Rev 3.3
% 2/18/06
%
% Modified by Tapashree Guha: March 2009.

```

```
function [OUT,varargout]=go_cal_psi(functname,D,varargin)
rand('state',sum(100*clock));
if nargin < 2
error('Not enough arguments.');
end

% PSO PARAMETERS
if nargin == 2      % only specified functname and D
VRmin=ones(D,1)*-100;
VRmax=ones(D,1)*100;
VR=[VRmin,VRmax];
minmax = 0;
P = [];
mv = 4;
plotfcn='goplotpso';
elseif nargin == 3  % specified functname, D, and mv
VRmin=ones(D,1)*-100;
VRmax=ones(D,1)*100;
VR=[VRmin,VRmax];
minmax = 0;
mv=varargin{1};
if isnan(mv)
mv=4;
end
P = [];
plotfcn='goplotpso';
elseif nargin == 4  % specified functname, D, mv, Varrange
mv=varargin{1};
if isnan(mv)
mv=4;
end
VR=varargin{2};
minmax = 0;
P = [];
plotfcn='goplotpso';
elseif nargin == 5  % Functname, D, mv, Varrange, and minmax
mv=varargin{1};
if isnan(mv)
mv=4;
end
VR=varargin{2};
minmax=varargin{3};
P = [];
plotfcn='goplotpso';
elseif nargin == 6  % Functname, D, mv, Varrange, minmax, and psoparams
mv=varargin{1};
if isnan(mv)
mv=4;
end
VR=varargin{2};
minmax=varargin{3};
P = varargin{4}; % psoparams
plotfcn='goplotpso';
elseif nargin == 7
% Functname, D, mv, Varrange, minmax, and psoparams, plotfcn
mv=varargin{1};
if isnan(mv)
mv=4;
end
VR=varargin{2};
minmax=varargin{3};
P = varargin{4}; % psoparams
plotfcn = varargin{5};
elseif nargin == 8
% Functname,D,mv,Varrange,minmax,and psoparams,plotfcn,PSOseedValue
mv=varargin{1};
if isnan(mv)
mv=4;
```

```
125 end
126 VR=varargin{2};
127 minmax=varargin{3};
128 P = varargin{4}; % psoparams
129 plotfcn = varargin{5};
130 PSOseedValue = varargin{6};
131 else
132 error('Wrong # of input arguments.');
133 end
134
135 % sets up default pso params
136 Pdef = [100 2000 24 2 2 0.9 0.4 1500 1e-25 250 NaN 0 0];
137 Plen = length(P);
138 P    = [P,Pdef(Plen+1:end)];
139
140 df      = P(1);
141 me      = P(2);
142 ps      = P(3);
143 ac1     = P(4);
144 ac2     = P(5);
145 iw1     = P(6);
146 iw2     = P(7);
147 iwe     = P(8);
148 ergrd   = P(9);
149 ergrdep = P(10);
150 errgoal = P(11);
151 trelea  = P(12);
152 PSOseed = P(13);
153
154
155 % set plotting flag
156 if (P(1))~=0
157 plotflg=1;
158 else
159 plotflg=0;
160 end
161
162 % preallocate variables for speed up
163 tr = ones(1,me)*NaN;
164
165 % take care of setting max velocity and position params here
166 if length(mv)==1
167 velmaskmin = -mv*ones(ps,D);     % min vel, psXD matrix
168 velmaskmax = mv*ones(ps,D);      % max vel
169 elseif length(mv)==D
170 velmaskmin = repmat(forcerow(-mv),ps,1); % min vel
171 velmaskmax = repmat(forcerow( mv),ps,1); % max vel
172 else
173 error('Max vel must be a scalar or samelength as prob dimension D');
174 end
175 posmaskmin  = repmat(VR(1:D,1)',ps,1);  % min pos, psXD matrix
176 posmaskmax  = repmat(VR(1:D,2)',ps,1);  % max pos
177 posmaskmeth = 3; % 3=bounce method
178
179 % PLOTTING
180 % fid_mopso=fopen('mospo_iteration_result.txt', 'wt');
181 % message = sprintf('PSO: %%g/%g iterations, GBest = %%20.20g.\n',me);
182 %fprintf(fid_mopso, 'PSO: %%g/%g iterations, GBest = %%20.20g.\n',me);
183
184 % INITIALIZE INITIALIZE INITIALIZE INITIALIZE INITIALIZE INITIALIZE
185
186 % initialize population of particles and their velocities at time zero,
187 % format of pos= (particle#, dimension)
188 % construct random population positions bounded by VR
189 pos(1:ps,1:D) = normmat(rand([ps,D]),VR',1);
190
191 % construct initial random velocities between -mv,mv
192 vel(1:ps,1:D) = normmat(rand([ps,D]),...
```

```
[forcecol(-mv),forcecol(mv)]',1);

% initial pbest positions vals
pbest = pos;

out = feval(functname,pos);
% returns column of match score values (1 for each particle)
%------------------------------

pbestval=out;   % initially, pbest is same as pos

% assign initial gbest here also (gbest and gbestval)
if minmax==1
% this picks gbestval when we want to maximize the function
[gbestval,idx1] = max(pbestval);
elseif minmax==0
% this works for straight minimization
[gbestval,idx1] = min(pbestval);
end

% preallocate a variable to keep track of gbest for all iters
bestpos        = zeros(me,D+1)*NaN;
gbest          = pbest(idx1,:);  % this is gbest position

bestpos(1,1:D) = gbest;

sentryval = gbestval;
sentry    = gbest;

% INITIALIZE END INITIALIZE END INITIALIZE END INITIALIZE END
%------------------------------------------------------------


rstflg = 0; % for dynamic environment checking
% start PSO iterative procedures
cnt    = 0;
% counter used for updating display according to df in the options
cnt2   = 0;
% counter used for the stopping subroutine based on error convergence
iwt(1) = iw1;
for i=1:me  % start epoch loop (iterations)

out        = feval(functname,[pos;gbest]);
outbestval = out(end,:);
out        = out(1:end-1,:);

tr(i+1)          = gbestval;
% keep track of global best val
te               = i;
% returns epoch number to calling program when done
bestpos(i,1:D+1) = [gbest,gbestval];

%assignin('base','bestpos',bestpos(i,1:D+1));
%------------------------------------------------------------------------
% this section does the plots during iterations
if plotflg==1
if (rem(i,df) == 0 ) || (i==me) || (i==1)
g=1-gbestval/10;
%fprintf(fid_mopso, message,i,g);
cnt = cnt+1; % count how many times we display (useful for movies)

eval(plotfcn); % defined at top of script

end  % end update display every df if statement
end % end plotflg if statement


% check for an error space that changes wrt time/iter
```

```
% threshold value that determines dynamic environment
% sees if the value of gbest changes more than some threshold value
% for the same location
chkdyn = 1;
rstflg = 0; % for dynamic environment checking

if chkdyn==1
threshld = 0.05;  % percent current best is allowed to change, .05 = 5% etc
letiter  = 5;
% # of iterations before checking environment, leave at
% least 3 so PSO has time to converge
outorng  = abs( 1- (outbestval/gbestval) ) > threshld;
samepos  = (max( sentry == gbest ));

if (outorng && samepos) && rem(i,letiter)==0
rstflg=1;
pbest     = pos; % reset personal bests to current positions
pbestval  = out;
vel       = vel*10; % agitate particles a little (or a lot)

% recalculate best vals
if minmax == 1
    [gbestval,idx1] = max(pbestval);
elseif minmax==0
%%%Implement Crowding Distance when selecting gbest, sort the pbests, and
%%%then select the gbest from the top 20%
    n_n=length(pbestval(:,1));
    pbestval_s=pbestval;
    pbestval_s=sort(pbestval_s);
    q_n=n_n/5;
    r_n=fix(1+(q_n-1)*rand(1));
    gbestval = pbestval_s(r_n);
    for i_n=1:n_n
        if(pbestval(i_n)==gbestval)
               idx1=i_n;
               break
        end
    end
    %[gbestval,idx1] = min(pbestval);
elseif minmax==2 % this section needs work
    [temp,idx1] = min((pbestval-ones(size(pbestval))*errgoal).^2);
    gbestval    = pbestval(idx1);
end

gbest  = pbest(idx1,:);

end  % end if outorng

sentryval = gbestval;
sentry    = gbest;

end % end if chkdyn

% find particles where we have new pbest, depending on minmax choice
% then find gbest and gbestval
%[size(out),size(pbestval)]
if rstflg == 0
if minmax == 0 %minimization problem
[tempi]            = find(pbestval>out); % new min pbestvals
pbestval(tempi,1)  = out(tempi);   % update pbestvals
pbest(tempi,:)     = pos(tempi,:); % update pbest positions
%%%Implement Crowding Distance when selecting gbest,
% sort the solutions based on their personal best, and
%%%then select the gbest from the top 20%
n_n1=length(pbestval(:,1));
pbestval_c=pbestval;
pbestval_c=sort(pbestval_c);
q_n1=n_n1/5;
```

```
r_n1=fix(1+(q_n1-1)*rand(1));
iterbestval = pbestval_c(r_n1);
for i_i=1:n_n1
    if(pbestval(i_i)==iterbestval)
        idx1=i_i;
        break
    end
end
%[iterbestval,idx1] = min(pbestval);

if gbestval >= iterbestval
    gbestval = iterbestval;
    gbest    = pbest(idx1,:);
end
elseif minmax == 1 %maximization problem
[tempi,dum]        = find(pbestval<=out); % new max pbestvals
pbestval(tempi,1)  = out(tempi,1); % update pbestvals
pbest(tempi,:)     = pos(tempi,:); % update pbest positions

[iterbestval,idx1] = max(pbestval);
if gbestval <= iterbestval
    gbestval = iterbestval;
    gbest    = pbest(idx1,:);
 end
end
end


%PSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSO

% get new velocities, positions (this is the heart of the PSO algorithm)
% each epoch get new set of random numbers
rannum1 = rand([ps,D]); % for Trelea and Clerc types
rannum2 = rand([ps,D]);
if     trelea == 2 %used for this research
%parameter set 2
vel = 0.729.*vel...                                % prev vel
+1.494.*rannum1.*(pbest-pos)...             % independent
+1.494.*rannum2.*(repmat(gbest,ps,1)-pos); % social
elseif trelea == 1
% parameter set 1
vel = 0.600.*vel...                                % prev vel
+1.700.*rannum1.*(pbest-pos)...             % independent
+1.700.*rannum2.*(repmat(gbest,ps,1)-pos); % social

end

% limit velocities here using masking
vel = ( (vel <= velmaskmin).*velmaskmin ) + ( (vel > velmaskmin).*vel );
vel = ( (vel >= velmaskmax).*velmaskmax ) + ( (vel < velmaskmax).*vel );

% update new position (PSO algo)
pos = pos + vel;

% position masking, limits positions to desired search space
% method: 0) no position limiting, 1) saturation at limit,
%         2) wraparound at limit , 3) bounce off limit
minposmask_throwaway = pos <= posmaskmin;  % these are psXD matrices
minposmask_keep      = pos >  posmaskmin;
maxposmask_throwaway = pos >= posmaskmax;
maxposmask_keep      = pos <  posmaskmax;

if     posmaskmeth == 1
% this is the saturation method
pos = ( minposmask_throwaway.*posmaskmin ) + ( minposmask_keep.*pos );
pos = ( maxposmask_throwaway.*posmaskmax ) + ( maxposmask_keep.*pos );
elseif posmaskmeth == 2
% this is the wraparound method
```

```
pos = ( minposmask_throwaway.*posmaskmax ) + ( minposmask_keep.*pos );
pos = ( maxposmask_throwaway.*posmaskmin ) + ( maxposmask_keep.*pos );
elseif posmaskmeth == 3
% this is the bounce method, particles bounce off the boundaries with -vel
pos = ( minposmask_throwaway.*posmaskmin ) + ( minposmask_keep.*pos );
pos = ( maxposmask_throwaway.*posmaskmax ) + ( maxposmask_keep.*pos );

vel = (vel.*minposmask_keep) + (-vel.*minposmask_throwaway);
vel = (vel.*maxposmask_keep) + (-vel.*maxposmask_throwaway);
else
% no change, this is the original Eberhart, Kennedy method,
% it lets the particles grow beyond bounds if psoparams (P)
% especially Vmax, aren't set correctly
end

%PSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSO
% check for stopping criterion based on speed of convergence to desired
% error
tmp1 = abs(tr(i) - gbestval);
if tmp1 > ergrd
cnt2 = 0;
elseif tmp1 ≤ ergrd
cnt2 = cnt2+1;
if cnt2 ≥ ergrdep
if plotflg == 1
    eval(plotfcn);
end
break
end
end

% this stops if using constrained optimization and goal is reached
if ¬isnan(errgoal)
if ((gbestval≤errgoal) && (minmax==0)) || ((gbestval≥errgoal) && (minmax==1))

if plotflg == 1
    fprintf(message,i,gbestval);
    disp(' ');
    disp('--> Error Goal reached, successful termination!');

    eval(plotfcn);
end
break
end

% this is stopping criterion for constrained from both sides
if minmax == 2
if ((tr(i)<errgoal) && (gbestval≥errgoal)) || ((tr(i)>errgoal) ...
        && (gbestval ≤ errgoal))
    if plotflg == 1
        fprintf(message,i,gbestval);
        disp(' ');
        disp('--> Error Goal reached, successful termination!');

        eval(plotfcn);
    end
    break
end
end % end if minmax==2
end  % end ¬isnan if

end  % end epoch loop

%% clear temp outputs
% evalin('base','clear temp_pso_out temp_te temp_tr;');

% output & return
OUT=[gbest';gbestval];
```

```
varargout{1}=[1:te]; %#ok<NBRAK>
varargout{2}=[tr(find(¬isnan(tr)))]; %#ok<FNDSB,NBRAK>

return
```

```
% forcecol.m
% function to force a vector to be a single column
%
%
% Brian Birge
% Rev 1.0
% 7/1/98


function[out]=forcecol(in)
len=prod(size(in));
out=reshape(in,[len,1]);
```

```
% forcerow.m
% function to force a vector to be a single row
%
% Brian Birge
% Rev 1.0
% 7/1/98


function[out]=forcerow(in)
len=prod(size(in));
out=reshape(in,[1,len]);
```

```
% goplotpso.m
%
% default plotting script used in PSO functions
%
% Brian Birge
% Rev 2.0
% 3/1/06
% Modified by Tapashree Guha March 2009

% setup figure, change this for your own machine
clf
set(gcf,'Position',[651    31   626   474]);
% this is the computer dependent part
%set(gcf,'Position',[743    33   853   492]);
set(gcf,'Doublebuffer','on');

% particle plot, upper right
subplot('position',[.7,.6,.27,.32]);
set(gcf,'color','k')

plot3(pos(:,1),pos(:,D),out,'b.','Markersize',7)

hold on
plot3(pbest(:,1),pbest(:,D),pbestval,'g.','Markersize',7);
plot3(gbest(1),gbest(D),gbestval,'r.','Markersize',25);

% crosshairs
offx = max(abs(min(min(pbest(:,1)),min(pos(:,1)))),...
    abs(max(max(pbest(:,1)),max(pos(:,1)))));

offy = max(abs(min(min(pbest(:,D)),min(pos(:,D)))),...
    abs(min(max(pbest(:,D)),max(pos(:,D)))));
plot3([gbest(1)-offx;gbest(1)+offx],...
    [gbest(D);gbest(D)],...
    [gbestval;gbestval],...
    'r-.');
plot3([gbest(1);gbest(1)],...
```

```
38      [gbest(D)-offy;gbest(D)+offy],...
39      [gbestval;gbestval],...
40      'r-.');
41
42  hold off
43
44  xlabel('Dimension 1','color','y')
45  ylabel(['Dimension ',num2str(D)],'color','y')
46  zlabel('Cost','color','y')
47
48  title('Particle Dynamics','color','w','fontweight','bold')
49
50  set(gca,'Xcolor','y')
51  set(gca,'Ycolor','y')
52  set(gca,'Zcolor','y')
53  set(gca,'color','k')
54
55  % camera control
56  view(2)
57  try
58      axis([gbest(1)-offx,gbest(1)+offx,gbest(D)-offy,gbest(D)+offy]);
59  catch
60      axis([VR(1,1),VR(1,2),VR(D,1),VR(D,2)]);
61  end
62
63  % error plot, left side
64  subplot('position',[0.1,0.1,.475,.825]);
65  semilogy(tr(find(~isnan(tr))),'color','m','linewidth',2)
66  %plot(tr(find(~isnan(tr))),'color','m','linewidth',2)
67  xlabel('No. of Iterations','color','y')
68  ylabel('Average Match Score','color','y')
69
70  g=1-gbestval/10;
71  if D==1
72      titstr1=sprintf(['Gbest Value:%11.6g = %s( [ %9.6g ] )'],...
73          g,strrep(functname,'_','\_'),gbest(1));
74  elseif D==2
75      titstr1=sprintf(['Gbest Value:%11.6g = %s( [ %9.6g, %9.6g ] )'],...
76          g,strrep(functname,'_','\_'),gbest(1),gbest(2));
77  elseif D==3
78      titstr1=sprintf(['Gbest Value:%11.6g = %s( [ %9.6g, %9.6g, %9.6g ] )'],...
79          g,strrep(functname,'_','\_'),gbest(1),gbest(2),gbest(3));
80  else
81      titstr1=sprintf(['Gbest Value:%11.6g = %s( [ %g inputs ] )'],...
82          g,strrep(functname,'_','\_'),D);
83  end
84  title(titstr1,'color','m','fontweight','bold');
85
86  grid on
87  %  axis tight
88
89  set(gca,'Xcolor','y')
90  set(gca,'Ycolor','y')
91  set(gca,'Zcolor','y')
92  set(gca,'color','k')
93
94  set(gca,'YMinorGrid','off')
95
96  % text box in lower right
97  % doing it this way so I can format each line any way I want
98  subplot('position',[.62,.1,.29,.4]);
99  clear titstr
100 if trelea==0
101     PSOtype  = 'Common PSO';
102     xtraname = 'Inertia Weight : ';
103     xtraval  = num2str(iwt(length(iwt)));
104
105 elseif trelea==2 %used for this research
```

```
    PSOtype  = ('MOPSO');
    xtraname = ' ';
    xtraval  = ' ';

elseif trelea==1

    PSOtype  = (['Trelea Type ',num2str(trelea)]);
    xtraname = ' ';
    xtraval  = ' ';
end
if isnan(errgoal)
    errgoalstr=' ';
else
    errgoalstr=num2str(errgoal);
end
if minmax==1
    minmaxstr = ['Maximization Problem'];
elseif minmax==0
    minmaxstr = ['Minimization Problem'];
end

if rstflg==1
    rststat1 = 'Environment Change';
    rststat2 = ' ';
else
    rststat1 = ' ';
    rststat2 = ' ';
end

titstr={'PSO Model: '      ,PSOtype;...
    'Dimensions : '    ,num2str(D);...
    '# of particles : ',num2str(ps);...
    minmaxstr          ,errgoalstr;...
    'Function : '      ,strrep(functname,'_','\_');...
    xtraname           ,xtraval;...
    rststat1           ,rststat2};

text(.1,1,[titstr{1,1},titstr{1,2}],'color','g','fontweight','bold');
hold on
text(.1,.9,[titstr{2,1},titstr{2,2}],'color','m');
text(.1,.8,[titstr{3,1},titstr{3,2}],'color','m');
text(.1,.7,[titstr{4,1}],'color','w');
text(.55,.7,[titstr{4,2}],'color','m');
text(.1,.6,[titstr{5,1},titstr{5,2}],'color','m');
text(.1,.5,[titstr{6,1},titstr{6,2}],'color','w','fontweight','bold');
text(.1,.4,[titstr{7,1},titstr{7,2}],'color','r','fontweight','bold');

legstr = {'Green = Personal Bests';...
    'Blue  = Current Positions';...
    'Red   = Global Best'};
text(.1,0.025,legstr{1},'color','g');
text(.1,-.05,legstr{2},'color','b');
text(.1,-.125,legstr{3},'color','r');

hold off

set(gca,'color','k');
set(gca,'visible','off');


%drawnow
```

```
% normmat.m
% takes a matrix and reformats the data to fit between a new range
%
% Brian Birge
% Rev 2.1
% 3/16/06 - changed name of function to avoid same name in robust control
```

```
% toolbox


function [out,varargout]=normmat(x,newminmax,flag)

if flag==0

  a=min(min((x)));
  b=max(max((x)));
  if abs(a)>abs(b)
     large=a;
     small=b;
  else
     large=b;
     small=a;
  end
  temp=size(newminmax);
  if temp(1)~=1
     error('Error: for method=0, range vector must be a 2' ...
     'element row vector');
  end
  den=abs(large-small);
  range=newminmax(2)-newminmax(1);
  if den==0
     out=x;
  else
     z21=(x-a)/(den);
     out=z21*range+newminmax(1)*ones(size(z21));
  end

%-------------------------------------------------------------
elseif flag==1
 a=min(x,[],1);
 b=max(x,[],1);
  for i=1:length(b)
     if abs(a(i))>abs(b(i))
        large(i)=a(i);
        small(i)=b(i);
     else
        large(i)=b(i);
        small(i)=a(i);
     end
  end
  den=abs(large-small);
  temp=size(newminmax);
  if temp(1)*temp(2)==2
     newminmaxA(1,:)=newminmax(1).*ones(size(x(1,:)));
     newminmaxA(2,:)=newminmax(2).*ones(size(x(1,:)));
  elseif temp(1)>2
     error('Error: for method=1, range matrix must have 2 rows and same'
     ..., 'columns as input matrix');
  else
     newminmaxA=newminmax;
  end

  range=newminmaxA(2,:)-newminmaxA(1,:);
  for j=1:length(x(:,1))
     for i=1:length(b)
        if den(i)==0
           out(j,i)=x(j,i);
        else
           z21(j,i)=(x(j,i)-a(i))./(den(i));
           out(j,i)=z21(j,i).*range(1,i)+newminmaxA(1,i);
        end
     end
  end
%-------------------------------------------------------------
elseif flag==2
```

```
   a=min(x,[],2);
   b=max(x,[],2);
   for i=1:length(b)
      if abs(a(i))>abs(b(i))
         large(i)=a(i);
         small(i)=b(i);
      else
         large(i)=b(i);
         small(i)=a(i);
      end
   end
   den=abs(large-small);
   temp=size(newminmax);
   if temp(1)*temp(2)==2
      newminmaxA(:,1)=newminmax(1).*ones(size(x(:,1)));
      newminmaxA(:,2)=newminmax(2).*ones(size(x(:,1)));
   elseif temp(2)>2
      error('Error: for method=2, range matrix must have 2 columns and' ...
          'same rows as input matrix');
   else
      newminmaxA=newminmax;
   end

   range=newminmaxA(:,2)-newminmaxA(:,1);
   for j=1:length(x(1,:))
      for i=1:length(b)
         if den(i)==0
            out(i,j)=x(i,j);
         else
            z21(i,j)=(x(i,j)-a(i))./([forcecol(den(i))]);
            out(i,j)=z21(i,j).*range(i,1)+newminmaxA(i,1);
         end
      end
   end

end
%-------------------------------------------------------------
varargout{1}=a;
varargout{2}=b;

return
```

```
%function to get the services when the req-srv pair is unequal.
%
% Created by Tapashree Guha March 2009
function [S]=getSrv(req,srv)
global A B %#ok<NUSED>
r=req;
s=srv;
temp_s=zeros(s,1);
S=zeros(r,5);
for i=1:s
      temp_s(i,1)= (-B(i,1)-B(i,2)+B(i,3)+B(i,4)+B(i,5));
end
[f_f idx_s]=sort(temp_s,1);
j=1;
while s>(req-1)
        k=idx_s(s);
        S(j,:)=B(k,:);
        s=s-1;
        j=j+1;
end
S=S(1:r,:);
return
```

```
%function to get the number of requesters for unequal number of req-srv
%pairs.
```

```
%
% Created by Tapashree Guha March 2009
function [R]= getReq(req,srv)
global A B %#ok<NUSED>
r=req;
s=srv;
temp=zeros(r,1);
R=zeros(s,5);

for i=1:r
      temp(i,1)= (-A(i,1)-A(i,2)+A(i,3)+A(i,4)+A(i,5));
end
[f idx]=sort(temp,1);
j=1;
while r>(srv-1)
        k=idx(r);
        R(j,:)=A(k,:);
        r=r-1;
        j=j+1;
end
R=R(1:s,:);
return
```

```
%particle_size_eval.m
%
% Keeping all other parameters constant, it sees the effect of the change
% in the particle size on the execution time and the match score for the
% PSO algorithm.
%
% Created By Tapashree Guha: April 2009.
function particle_size_eval()
clear
clc
num_part=25;
num_part_id=zeros(20,1);
match_score=zeros(20,1);
time_pso=zeros(20,1);
fid_varpart=fopen('particle_eval.txt','a');
i=1;
while num_part≤500
    fprintf('%d particles now!\n\n', num_part)
    [T,inp,M]=var_particle_psi(num_part,500,500);
    time_pso(i,1)=T;
    match_score(i,1)=M;
    num_part_id(i,1)=num_part;
    fprintf(fid_varpart,'FOR PARTICLE SIZE %d\n**********\n', num_part);
    fprintf(fid_varpart,'The execution time is, %f\n', time_pso(i,1));
    fprintf(fid_varpart,'The Match Score is, %f\n', match_score(i,1));
    num_part=num_part+25;
    i=i+1;
end

figure (2)
plot(num_part_id, match_score,'--*b','LineWidth',2,...
                              'MarkerEdgeColor','k',...
                              'MarkerFaceColor','g',...
                              'MarkerSize',8)
ylim([0 1])
xlabel('No of Particles.')
ylabel('Average Match Score')
title('Effect of Particle Size on Average Match Score.');

figure (3)
plot(num_part_id, time_pso,'--*g','LineWidth',2,...
                              'MarkerEdgeColor','k',...
                              'MarkerFaceColor','g',...
                              'MarkerSize',8)
%ylim([0 1])
```

```
xlabel('No of Particles.')
ylabel('Execution Time (sec).')
title('Effect of Particle Size on Execution Time.');
```

```
% same as psi_def.m just the particle size are made variable. It works
% with the function particle_size_eval to evaluate the effect of particle
% size on time and match score of the MOPSO algorithm.
%
% Created By: Tapashree Guha, April 2009.

function [e,inP,MS]=var_particle_psi(n, req,srv)
format long
global A B qosAtt rangeP
if(nargin<1)
    error('Not Enough input arguments');
else
    t=cputime;
    %Initializing Variables
    x=req; y=srv;
    S=rand(y,5); R=rand(x,5);
    A=rand(req,5); %Requesters
    B=rand(srv,5); %Services

    % writing the Service – Requester QoS Matric to Files, also the Matchscore
    % to a file.
    fid = fopen('service.txt', 'wt');
    fid2 = fopen('requester.txt', 'wt');
    fprintf(fid, '%f  %f  %f  %f %f\n', A);
    fprintf(fid2, '%f  %f  %f  %f  %f\n', B);
    fclose(fid);
    fclose(fid2);

    if req<srv
        R=A;
        S=getSrv(req,srv);
        qosAtt=[R S];
        %disp(S);
    else if req>srv
            R=getReq(req,srv);
            S=B;
            qosAtt=[R S];
            %disp(R)
        else if req==srv
                R=A;
                S=B;
                qosAtt=[R S];

            end
        end
    end
    %getting dimenstions of A and B
    a=length(R(:,1));
    b=length(S(:,1));

    P=fix(1+(10-1)*rand(a,2));

    rangeP=sort(P,2);
    for i=1:a
        if (rangeP(i,1)==rangeP(i,2)&& rangeP(i,1)~=1)
            rangeP(i,1)=rangeP(i,1)-1;
        else if (rangeP(i,1)==rangeP(i,2) && rangeP(i,1)==1)
                rangeP(i,2)=rangeP(i,2)+1;
            end
        end
    end

    dim=length(qosAtt(:,1)); %dimension
    Pdef = [50 5000 n 2 2 0.5 0.3 1500 1e-25 150 NaN 2 0]; %PSO Parameters
```

```
    [OUT]=go_cal_psi('obj_cal_psi',dim,1,rangeP,0,Pdef); % Global Bests
    out=abs(OUT);
    inP=out(1:dim);

    [F]=obj_cal_psi(inP');
    MS=1-(F/10);
    e=cputime-t;
end
```

```
% similar_psi_def.m
%
% A function that defines the Input parameters, such as the requesters,
% services, for the go_cal_psi function. And finally returns
% the execution time e, final average Match Score. In this case the
% function inputs the Similar Requester and Service Provider vectors.
%
% Created by Tapashree Guha: March 2009.
function [e,MS]=similar_psi_def(R_eq,S_rv)
format long
global A B qosAtt rangeP
if(nargin<1)
    error('Not Enough input arguments');
else
    t=cputime;

    A=R_eq; %Requesters
    B=S_rv; %Services

    req=length(A(:,1));
    srv=length(B(:,1));
    x=req; y=srv;
    S=rand(y,5); R=rand(x,5);

    % writing the Service - Requester QoS Matric to Files, also the
    % Matchscore to a file.
    fid = fopen('service.txt', 'wt');
    fid2 = fopen('requester.txt', 'wt');
    fprintf(fid2, '%f  %f  %f  %f %f\n', A);
    fprintf(fid, '%f  %f  %f  %f  %f\n', B);
    fclose(fid);
    fclose(fid2);

    if req<srv
        R=A;
        S=getSrv(req,srv);
        qosAtt=[R S];

    else if req>srv
            R=getReq(req,srv);
            S=B;
            qosAtt=[R S];

        else if req==srv
                R=A;
                S=B;
                qosAtt=[R S];

            end
        end
    end
    %getting dimenstions of A and B
    a=length(R(:,1));
    b=length(S(:,1));


    P=fix(1+(10-1)*rand(a,2));

    rangeP=sort(P,2);
```

```
60      for i=1:a
61          if (rangeP(i,1)==rangeP(i,2)&& rangeP(i,1)≠1)
62              rangeP(i,1)=rangeP(i,1)-1;
63          else if (rangeP(i,1)==rangeP(i,2) && rangeP(i,1)==1)
64                  rangeP(i,2)=rangeP(i,2)+1;
65              end
66          end
67      end
68
69      dim=length(qosAtt(:,1)); %dimension
70      Pdef = [50 5000 25 2 2 0.5 0.3 1500 1e-25 150 NaN 2 0]; %PSO Parameters
71      [OUT]=go_cal_psi('obj_cal_psi',dim,1,rangeP,1,Pdef); % Global Bests
72      out=abs(OUT);
73      inP=out(1:dim);
74
75      [F]=obj_cal_psi(inP');
76      MS=1-(F/10);
77      e=cputime-t;
78
79  end
```

A.3 CSS Algorithm Code

This section gives the source code for the CSS algorithm.

```
%go_cal_css.m
%           Implements the proposed CSS algorithm.
%            Calculates the overall match score, based on the
%            requester–services QoS match value pair. Takes in 3 arguments:
%            1) The list of requesters
%            2) The list of available services
%            The match value for each QoS parameter for each R–S pair is
%            calculated. Then the match score is calcuated for each R–S
%            pair and is stored in the MS matrix.
%
%
% Created by: Tapashree Guha, February. 2009.



function [e,matchScore]=go_cal_css(req,srv)
global A_cs B_cs normMS MS
%Insert the values of the matchscores from the text file into a matrix
%and Initialize Variables.
MS=zeros(req,srv); normMS = zeros(req,srv); matchScore = 0; b_=1; temp =0;
fid_cs = fopen('requester.txt');
fid2_cs = fopen('service.txt');
A_cs  = fscanf(fid_cs, '%f', [req,5]);
B_cs  = fscanf(fid2_cs, '%f', [srv,5]);

fclose(fid_cs);
fclose(fid2_cs);
[A_r A_c]=size(A_cs);
[B_r B_c]=size(B_cs);


a=length(A_cs(:,1));
b=length(B_cs(:,1));

matchScore = 0; b1=1; temp =0; t=0;
fid4 = fopen('matchscores.txt', 'wt');
t=cputime;  %Calculates the CPU Time

%%%%%%%%% The Match Value Calculation of the Requester – Service Pair %%%%%
for r = 1:a
for r1 = 1:b
    c = 1;
    for c1 = 1:5
        matchvalTemp = abs(1–((abs(A_cs(r,c)–B_cs(r1,c1)))/A_cs(r,c)));
        %adding all the match value for individual
        %Service–Requester Attribute pair to get the Matchscores.
        matchScore = matchScore+matchvalTemp;
        c = c+1;
    end
    MS(r,r1) = matchScore/5; % Storing the matchscore in MS matrix
    %fprintf(fid4, '%f ',MS(r,r1));
    matchScore = 0;
end
%(fid4, '\n');
end

% Normalizing the Match Score Matrix MS
maxVal = max(max(MS));
for rv = 1:r
for cv = 1:r1
    normMS(rv,cv) = MS(rv,cv)/maxVal;
    fprintf(fid4, '%f ', normMS(rv,cv));
end
```

```
    fprintf(fid4, '\n');
end

%Finding the maximum match score from each row (unique to a column) ,
% and assigning the requester to corresponding service.
W = normMS; count=1;
max_ms = normMS(1,1);
if a>b || a==b
for i=1:b
for j=1:b
    if(W(i,j)>max_ms)
        max_ms=W(i,j);
        b_=j;
    end
end
%disp(max_ms)
temp=temp+max_ms;
if(i<a)
    for k=i+1:a
        W(k,b_)=0;
    end
    max_ms=W(i+1,1);
end
end

matchScore = temp/10;
else
for i=1:a
for j=1:a
    if(W(i,j)>max_ms)
        max_ms=W(i,j);
        b_=j;
    end
end

temp=temp+max_ms;
if(i<a)
    for k=i+1:a
        W(k,b_)=0;
    end
    max_ms=W(i+1,1);
end
end
matchScore = temp/10;
end
e = cputime-t;
fclose(fid4);
```

```
% go_plot_css_ms.m
% A plot function that plots and compares the avrage match score for the
% traditional algorithm. This function takes the data from text files:
% avg_match.txt % It plots the avg execution time and the match score for all
% the three scenarios: 1) 500-500, 2) 500-1000 3) 1000-500.
%
% Created By: Tapashree Guha, April 2009.
function go_plot_css_ms
clc
avg_m=fopen('avg_match.txt');
avgM = fscanf(avg_m, '%f', [2,3]);
avgM=avgM';
avgM=avgM(:,2);

figure(6)

bar(avgM,0.5,'m')
set(gca,'XTick',1:1:3)
set(gca,'XTickLabel',{'500-500','500-1000','1000-500'})
xlabel('Requester-Service Pairs')
```

```
ylabel('Average Match Scores.')
title 'Average Match Score for Requester—Service pairs of CSS.'
```

```
%Generates Similar requests to test the performance of the algorithm.
%
%
% Created By Tapashree Guha, April 2009
function similar_match_comp
clc
clear
r_r=500; s_s=500; n=5; k=1; l=1; m=1; z=1; o=1;
R_eq=rand(500,5); S_rv=rand(500,5);

while n≤100
    fprintf('Its %d percentage now!\n',n)
    n_id(k,1)=n;
    temp=(n/100*500);
    for i=1:temp
        R_eq(i,:)=0.4+(0.7—0.4)*rand(1,5);
    end

    n=n+10;
    [t1,M1]=similar_psi_def(R_eq,S_rv);
    [t2,M2]=go_cal_css(r_r,s_s);
    T1(l,1)=t1; T2(m,1)=t2;
    M_pso(z,1)=M1; M_mm(o,1)=M2;
    l=l+1; m=m+1; z=z+1; o=o+1; k=k+1;
end

fprintf('I am out!\n')

%Plot the time
figure(2)
plot(n_id,T1,'—-ro','LineWidth',2,...
                                'MarkerEdgeColor','k',...
                                'MarkerFaceColor','k',...
                                'MarkerSize',8)
hold on
plot(n_id,T2,'—-g*','LineWidth',2,...
                                'MarkerEdgeColor','k',...
                                'MarkerFaceColor','k',...
                                'MarkerSize',8)
hold off
h = legend('PSI','CSS',2, 'Location','NorthEast');
set(h,'Interpreter','none')
xlabel('Percentage of Similar Requests.')
ylabel('Execution time(ms)')
title 'Execution Times'

%plot the Match Scores
figure(3)
plot(n_id,M_pso, '—-ro','LineWidth',2,...
                                'MarkerEdgeColor','k',...
                                'MarkerFaceColor','k',...
                                'MarkerSize',8)
hold on
plot(n_id,M_mm,'—-g*','LineWidth',2,...
                                'MarkerEdgeColor','k',...
                                'MarkerFaceColor','k',...
                                'MarkerSize',8)
hold off
h = legend('PSI','CSS',2);
set(h,'Interpreter','none')
xlabel('Percentage of Similar Requests.')
ylabel('Average Match Score')
title 'Average MatchScore Values'
```

```
%algo_exec_eval.m
%An experimental Script to evaluate and analyze the execution times taken
%by the algorithms from 100-100 to 1000-1000 requester-service pairs.
%It calculates the execution times taken by both the algorithms in two
%separate text files and also plots them in two
%separate graphs. The PSI Parameters are fixed as follows:
%No. of particles:25
%Acceleration Constant 1: 2;
%Acceleration Constant 2: 2;
%Initial Inertial weight: 0.5;
%Final Inertial weight: 0.3;

%Created By: Tapashree Guha - April 2009
function algo_exec_eval
clc
clear
num_srv=100;
num_req=100;
pso_exec=zeros(10,1);
mm_exec=zeros(10,1);
num_req_id=zeros(10,1);
i=1;
fid_pso_exec=fopen('pso-exec-eval.txt','a');
fid_mm_exec=fopen('csmm-exec-eval.txt','a');
while num_req<=1000
        fprintf('Now R-S pair is %d\n',num_req)
        [t1,X,t2,Y]=run_psi_css(num_req,num_srv);
        pso_exec(i,1)=t1;
        mm_exec(i,1)=t2;
        num_req_id(i,1)=num_req;
        fprintf(fid_pso_exec,'The PSI execution time for %d-%d REQ-SRV',...
                'pair is: %f\n\n',num_req,num_srv,pso_exec(i,1));
        fprintf(fid_mm_exec,'The CSS execution time for %d-%d REQ-SRV', ...
                'pair is: %f\n\n',num_req,num_srv,mm_exec(i,1));
        num_req=num_req+100;
        num_srv=num_srv+100;
        i=i+1;

end
figure (2)
plot(num_req_id, pso_exec,'--*b','LineWidth',2,...
                                'MarkerEdgeColor','k',...
                                'MarkerFaceColor','g',...
                                'MarkerSize',8)
xlabel('Requester-Service Pair')
ylabel('Execution Time(ms)')
title('Average Execution Time of PSI algorithm');

figure (3)
plot(num_req_id,mm_exec,'--*g','LineWidth',2,...
                                'MarkerEdgeColor','k',...
                                'MarkerFaceColor','g',...
                                'MarkerSize',8)
xlabel('Requester-Service Pair')
ylabel('Execution Time(ms)')
title('Average Execution Time of CSS algorithm');

fclose(fid_pso_exec);
fclose(fid_mm_exec);
```

```
% go_plot_csmmVspso.m
% A plot function that plots and compares the traditional and PSO algorithm
% based on the execution time and the match scores output by both.
% This function takes the data from two text files: avg_match.txt and
% avg_exec.txt. It plots the avg execution time and the match score for all
% the three scenarios: 1) 500-500, 2) 500-1000 3) 1000-500.
%
% Created By: Tapashree Guha, April 2009.
```

```
function go_plot_css_Vs_psi
clc
avg_m=fopen('avg_match.txt');
avgM = fscanf(avg_m, '%f', [2,3]);
avgM=avgM';


avg_t=fopen('avg_exec.txt');
avgT = fscanf(avg_t, '%f', [2,3]);
avgT=avgT';

figure(4)

bar(avgM)
legend('PSI','CSS')
set(gca,'XTick',1:1:3)
set(gca,'XTickLabel',{'500-500','500-1000','1000-500'})
xlabel('Requester-Service Pairs')
ylabel('Average Match Scores.')
title 'Average Match Score for Requester-Service pairs of CSS and PSI.'

figure(5)
bar(avgT)
legend('PSI','CSS')
set(gca,'XTick',1:1:3)
set(gca,'XTickLabel',{'500-500','500-1000','1000-500'})
xlabel('Requester-Service Pairs')
ylabel('Average Execution Time(sec).')
title 'Average Execution Time of Requester-Service pairs of CSS and PSI.'
```

A.4 Munkres Algorithm Code

This section gives the source code for the Munkres algorithm. [2]

```
% munkres.m
% Takes in the number of requesters and service providers, and returns the
% average matching score, and the CPU Time taken to execute the algorithm.
%
% The algorithm basically works on the matchscore matrix that represents
% the matchscores of each requester—service pairs.
% Matching returns a MxN matrix with ones in the place where the requester
% and the services have been matched and zeros elsewhere.
% Match returns the average match scores of the maximum matching
%
% References:
%
% 1. The step by step explaination of the algorithm was found in:
% http://216.249.163.93/bob.pilgrim/445/munkres.html.
%
% 2. Algorithms for Assignment and Transportation Problems, James Munkres,
% Journal of the Society for Industrial and Applied Mathematics Volume 5,
% Number 1, March, 1957
%
% 3. An extension of the Munkres algorithm for the assignment problem to
% rectangular matrices.F. Burgeois and J.—C. Lasalle.,Communications of the
% ACM, 142302—806, 1971.
%
% Created by: Tapashree Guha March 2009.

function [e,match]=munkres(req,srv)
global normMS

%Insert the values of the matchscores from the text file into a matrix
%and Initialize Variables.
r=req; c=srv;

if c>r
Perf=normMS(:,1:r);
else
Perf= normMS(1:c,:);
end


Matching = zeros(size(Perf));

%*************************************************************************
% STEP 0: Make sure there are as many number of columns as there are
% rows. Find the number in each column that are connected, find the
% number in each row that are connected, find the columns(vertices) and
% rows(vertices) that are isolated, assemble condensed Perf Matrix.
%*************************************************************************

num_y = sum(¬isinf(Perf),1);
num_x = sum(¬isinf(Perf),2);
x_con = find(num_x≠0);
y_con = find(num_y≠0);
P_size = max(length(x_con),length(y_con));
P_cond = zeros(P_size);
P_cond(1:length(x_con),1:length(y_con)) = Perf(x_con,y_con);
if isempty(P_cond)
Cost = 0;
return
end

% Ensure that a perfect matching exists
```

[2] I would like to thank the author of [111], for providing the step-by-step breakdown of the Munkres Algorithm.

```
Edge = P_cond;
Edge(P_cond≠Inf) = 0;
cnum = min_line_cover(Edge);
Pmax = max(max(P_cond(P_cond≠Inf)));
P_size = length(P_cond)+cnum;
P_cond = ones(P_size)*Pmax;
P_cond(1:length(x_con),1:length(y_con)) = Perf(x_con,y_con);

%************************************************
% MAIN PROGRAM: CONTROLS WHICH STEP IS EXECUTED
%************************************************
%Calculates the CPU Time
t1=cputime;
exit_flag = 1;
stepnum = 1;
while exit_flag
switch stepnum
case 1
    [P_cond,stepnum] = step1(P_cond);
case 2
    [r_cov,c_cov,M,stepnum] = step2(P_cond);
case 3
    [c_cov,stepnum] = step3(M,P_size);
case 4
    [M,r_cov,c_cov,Z_r,Z_c,stepnum] = step4(P_cond,r_cov,c_cov,M);
case 5
    [M,r_cov,c_cov,stepnum] = step5(M,Z_r,Z_c,r_cov,c_cov);
case 6
    [P_cond,stepnum] = step6(P_cond,r_cov,c_cov);
case 7
    exit_flag = 0;
end
end

% Remove all the virtual satellites and targets and uncondense the
% Matching to the size of the original performance matrix.

Matching(x_con,y_con) = M(1:length(x_con),1:length(y_con));
if r<c
match = (sum(sum(Perf(Matching==1))))/c;
else
match = (sum(sum(Perf(Matching==1))))/r;
end

%fprintf('\n\n****RESULTS FOR MUNKRES ALGORITHM.****\n\n');
% fprintf('avg_match_munkres: %f.\n',match)

e = cputime-t1;
% fprintf('avg_exec_munkres : %f Seconds.\n', e)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   STEP 1: Find the largest number of zeros in each row
%           and subtract that maximum from its row
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [P_cond,stepnum] = step1(P_cond)

P_size = length(P_cond);

% Loop throught each row
for ii = 1:P_size
rmax = max(P_cond(ii,:));
P_cond(ii,:) = P_cond(ii,:)-rmax;
end

stepnum = 2;

%************************************************************************
```

```
%   STEP 2: Find a zero in P_cond. If there are no starred zeros in its
%           column or row start the zero. Repeat for each zero
%*************************************************************************

function [r_cov,c_cov,M,stepnum] = step2(P_cond)

% Define variables
P_size = length(P_cond);
r_cov = zeros(P_size,1);  % A vector that shows if a row is covered
c_cov = zeros(P_size,1);  % A vector that shows if a column is covered
M = zeros(P_size);
% A mask that shows if a position is starred or primed

for ii = 1:P_size
for jj = 1:P_size
if P_cond(ii,jj) == 0 && r_cov(ii) == 0 && c_cov(jj) == 0
    M(ii,jj) = 1;
    r_cov(ii) = 1;
    c_cov(jj) = 1;
end
end
end

% Re-initialize the cover vectors
r_cov = zeros(P_size,1);  % A vector that shows if a row is covered
c_cov = zeros(P_size,1);  % A vector that shows if a column is covered
stepnum = 3;

%*************************************************************************
%   STEP 3: Cover each column with a starred zero. If all the columns are
%           covered then the matching is maximum
%*************************************************************************

function [c_cov,stepnum] = step3(M,P_size)

c_cov = sum(M,1);
if sum(c_cov) == P_size
stepnum = 7;
else
stepnum = 4;
end

%*************************************************************************
%   STEP 4: Find a noncovered zero and prime it.  If there is no starred
%           zero in the row containing this primed zero, Go to Step 5.
%           Otherwise, cover this row and uncover the column containing
%           the starred zero. Continue in this manner until there are no
%           uncovered zeros left. Save the smallest uncovered value and
%           Go to Step 6.
%*************************************************************************
function [M,r_cov,c_cov,Z_r,Z_c,stepnum] = step4(P_cond,r_cov,c_cov,M)

P_size = length(P_cond);

zflag = 1;
while zflag
% Find the first uncovered zero
row = 0; col = 0; exit_flag = 1;
ii = 1; jj = 1;
while exit_flag
if P_cond(ii,jj) == 0 && r_cov(ii) == 0 && c_cov(jj) == 0
    row = ii;
    col = jj;
    exit_flag = 0;
end
jj = jj + 1;
if jj > P_size; jj = 1; ii = ii+1; end
if ii > P_size; exit_flag = 0; end
```

```
198 end
199
200 % If there are no uncovered zeros go to step 6
201 if row == 0
202 stepnum = 6;
203 zflag = 0;
204 Z_r = 0;
205 Z_c = 0;
206 else
207 % Prime the uncovered zero
208 M(row,col) = 2;
209 % If there is a starred zero in that row
210 % Cover the row and uncover the column containing the zero
211 if sum(find(M(row,:)==1)) ~= 0
212     r_cov(row) = 1;
213     zcol = find(M(row,:)==1);
214     c_cov(zcol) = 0;
215 else
216     stepnum = 5;
217     zflag = 0;
218     Z_r = row;
219     Z_c = col;
220 end
221 end
222 end
223
224 %*************************************************************************
225 % STEP 5: Construct a series of alternating primed and starred zeros as
226 %         follows.  Let Z0 represent the uncovered primed zero in Step 4.
227 %         Let Z1 denote the starred zero in the column of Z0 (if any).
228 %         Let Z2 denote the primed zero in the row of Z1 (there will always
229 %         be one).  Continue until the series terminates at a primed zero
230 %         that has no starred zero in its column.  Unstar each starred
231 %         zero of the series, star each primed zero of the series, erase
232 %         all primes and uncover every line in the matrix.  Go to Step 3.
233 %*************************************************************************
234
235 function [M,r_cov,c_cov,stepnum] = step5(M,Z_r,Z_c,r_cov,c_cov)
236
237 zflag = 1;
238 ii = 1;
239 while zflag
240 % Find the index number of the starred zero in the column
241 rindex = find(M(:,Z_c(ii))==1);
242 if rindex > 0
243 % Save the starred zero
244 ii = ii+1;
245 % Save the row of the starred zero
246 Z_r(ii,1) = rindex;
247 % The column of the starred zero is the same as the column of the
248 % primed zero
249 Z_c(ii,1) = Z_c(ii-1);
250 else
251 zflag = 0;
252 end
253
254 % Continue if there is a starred zero in the column of the primed zero
255 if zflag == 1;
256 % Find the column of the primed zero in the last starred zeros row
257 cindex = find(M(Z_r(ii),:)==2);
258 ii = ii+1;
259 Z_r(ii,1) = Z_r(ii-1);
260 Z_c(ii,1) = cindex;
261 end
262 end
263
264 % UNSTAR all the starred zeros in the path and STAR all primed zeros
265 for ii = 1:length(Z_r)
```

```
266 if M(Z_r(ii),Z_c(ii)) == 1
267 M(Z_r(ii),Z_c(ii)) = 0;
268 else
269 M(Z_r(ii),Z_c(ii)) = 1;
270 end
271 end
272
273 % Clear the covers
274 r_cov = r_cov.*0;
275 c_cov = c_cov.*0;
276
277 % Remove all the primes
278 M(M==2) = 0;
279
280 stepnum = 3;
281
282 % ************************************************************************
283 % STEP 6: Add the minimum uncovered value to every element of each covered
284 %         row, and subtract it from every element of each uncovered column.
285 %         Go to Step 4 without altering any stars, primes, or covered lines.
286 %************************************************************************
287
288 function [P_cond,stepnum] = step6(P_cond,r_cov,c_cov)
289 a = find(r_cov == 0);
290 b = find(c_cov == 0);
291 minval = min(min(P_cond(a,b)));
292
293 P_cond(find(r_cov == 1),:) = P_cond(find(r_cov == 1),:) + minval;
294 P_cond(:,find(c_cov == 0)) = P_cond(:,find(c_cov == 0)) - minval;
295
296 stepnum = 4;
297
298 function cnum = min_line_cover(Edge)
299
300 % Step 2
301 [r_cov,c_cov,M,stepnum] = step2(Edge);
302 % Step 3
303 [c_cov,stepnum] = step3(M,length(Edge));
304 % Step 4
305 [M,r_cov,c_cov,Z_r,Z_c,stepnum] = step4(Edge,r_cov,c_cov,M);
306 % Calculate the deficiency
307 cnum = length(Edge)-sum(r_cov)-sum(c_cov);
```

Appendix B

Screenshots of PSO Toolbox

This section shows the screen shots of the PSI selection algorithms generated by the MATLAB MOPSO toolbox for 500-500 requester-service pairs, at 50th iteration for 25 particles and 1000 particle respectively.

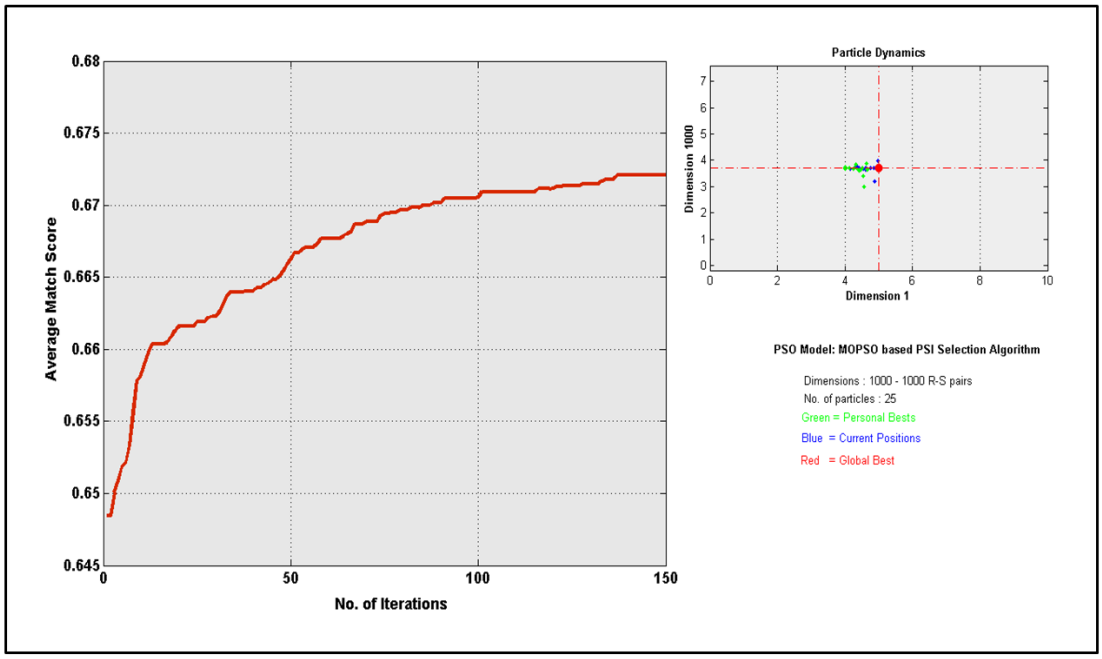

Figure B.1: Screenshot of PSI algorithm for 25 particles.

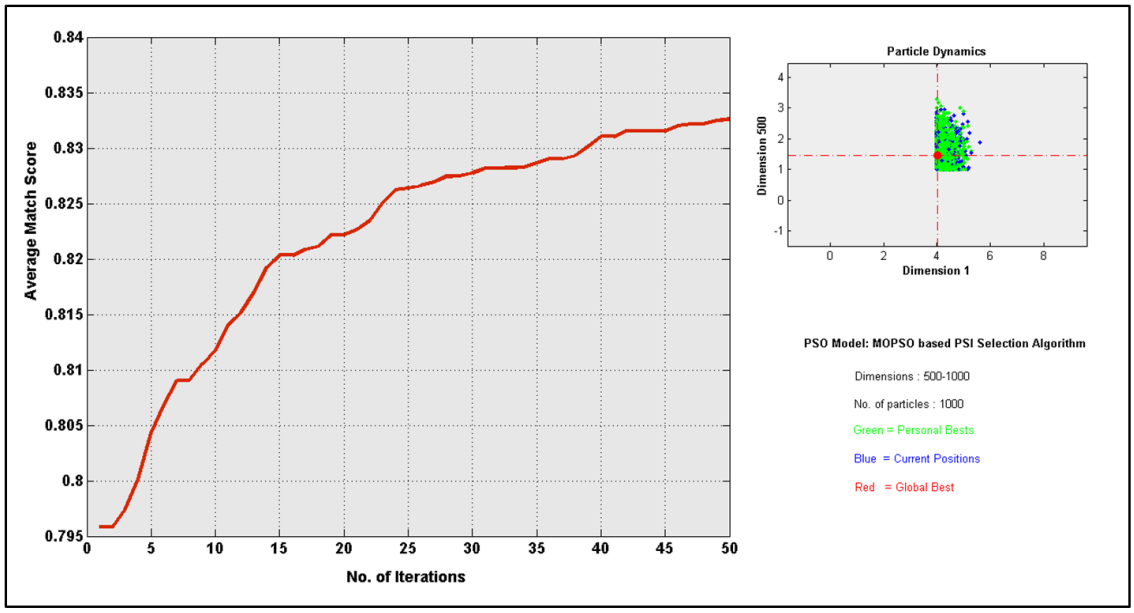

Figure B.2: Screenshot of PSI algorithm for 1000 particles.